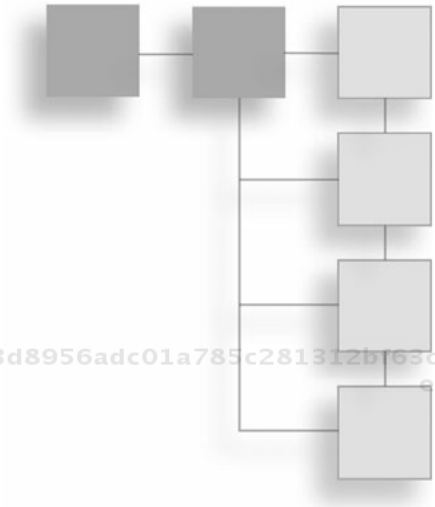
Chapter 7



PyQt

In the previous chapter you learned about file handling. You learned to open a file in different modes, read its contents, update existing content, delete content, append new content, and make a copy. You learned to read files sequentially as well as randomly. Besides this, you also learned to create binary files, pickle and unpickle objects, and implement exception handling.

The applications that you have created so far were console-based applications. From now on you will be learning to develop graphical user interface (GUI) applications in Python through PyQt. This chapter covers the following:

- Introduction to Qt toolkit and PyQt
- PyQt installation
- Window and dialogs
- Creating GUI Application through coding
- Using Qt Designer
- Understanding fundamental widgets—Label, Line Edit, and Push Button
- Event handling in PyQt
- First Application in Qt Designer
- Connecting to the predefined slots
- Using custom slots

- Converting data types
- Defining buddies and setting tab order

Let's begin the chapter with an introduction to Qt toolkit.

Qt Toolkit

Qt toolkit, known simply as Qt, is a cross-platform application and UI framework developed by Trolltech that is used for developing GUI applications. It runs on several platforms, including Windows, Mac OS X, Linux, and other UNIX platforms. It is also referred to as a *widget toolkit* because it provides widgets such as buttons, labels, text boxes, pushbuttons, and list boxes, which are required in designing a GUI. It includes a cross-platform collection of classes, integrated development tools, and a cross-platform IDE.

PyQt

PyQt is a set of Python bindings for the Qt toolkit. PyQt combines all the advantages of Qt and Python. With PyQt, you can include Qt libraries in Python code, enabling you to write GUI applications in Python. In other words, PyQt allows you to access all the facilities provided by Qt through the Python code. Since PyQt depends on the Qt libraries to run, when you install PyQt, the required version of Qt is also installed automatically on your machine.

Installing PyQt

You need to have Python Interpreter installed on your system before you install PyQt. Recall from Chapter 1, "Python and Its Features," that you have already installed Python 3.2 on your system, so you can go ahead and download PyQt from http://www.riverbankcomputing.co.uk/software/pyqt/download. The latest version at the time of this writing is PyQt version 4.8.5 for Python 3.2. The name of the downloaded file is `PyQt-Py3.2-x86-gpl-4.8.5-1.exe`. Just double-click the downloaded file to begin installation. The first screen that you see is a welcome screen to the PyQt Setup Wizard, as shown in Figure 7.1. The screen displays general information about the components that come with PyQt. Select Next to move forward.

Note

Your operating system may complain, saying the program is from an unknown publisher and may harm your computer. Select the Yes button to proceed with the installation. If you don't see a Yes button, select Actions to see the list of possible actions. In the dialog that appears, select the More Options drop-down and select Run to begin with the installation procedure.

Figure 7.1
PyQt Setup Wizard dialog.

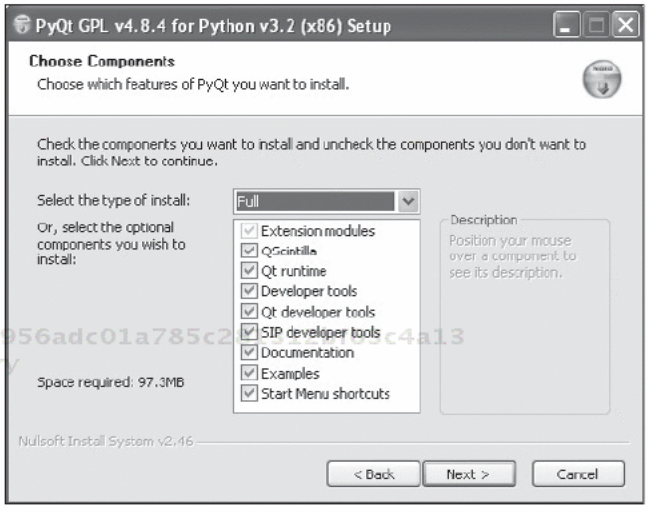


Figure 7.2
Selecting the features of PyQt to install.

The next screen shows the License Agreement, which you need to read and agree to before installing PyQt. Select I Agree to continue installation. Next, you get a screen that shows the list of components that you can install with PyQt (see Figure 7.2). You can select or deselect any component. The dialog also shows the disk space that will be required for installing the selected components.

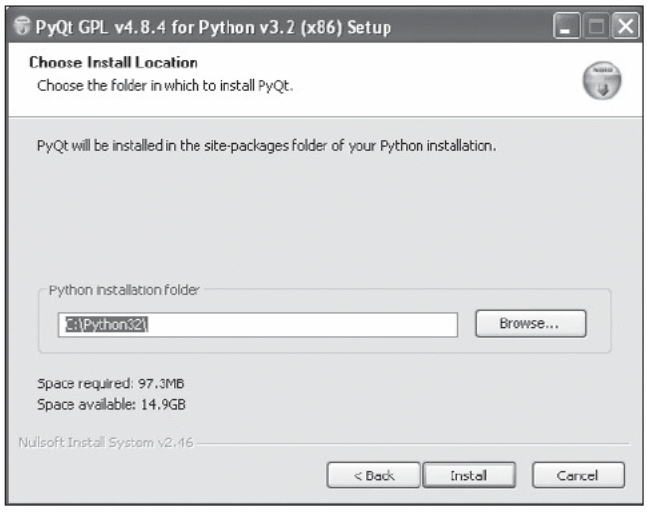


Figure 7.3
Specifying a location for PyQt installation.

Let's go ahead with Full Installation and select Next to move on. The next screen will prompt you to specify the name and location of the folder where Python 3.2 is installed. The reason is that PyQt is installed in the `site-packages` folder of the Python installation. The wizard auto-detects and shows the location of the Python installation by default, as shown in Figure 7.3. You can also select Browse to modify the folder name. After specifying the location of the Python installation, select Install to begin copying and installing the PyQt files.

When PyQt files are copied and installed, you will be prompted to select Finish to close the wizard.

Note

Don't forget to set the path of the PyQt folder so that you can access it from any folder on your computer.

Congratulations! You successfully installed PyQt on your computer. You can now begin creating your GUI applications. When doing so, you might be prompted to specify whether you want to create a main window application or a dialog application. What does this mean? Let's see.

Window and Dialogs

A GUI application may consist of a main window with several dialogs or just dialogs. A small GUI application usually consists of at least one dialog. A dialog application

contains buttons. It doesn't contain a menu bar, toolbar, status bar, or central widget, whereas a main window application normally has all of those. A central widget is one that contains other widgets.

Dialogs are of two types: *modal* and *modeless*. A modal dialog is one that blocks the user from interacting with other parts of the application. The dialog is the only part of the application that the user can interact with. Until the dialog is closed, no other part of the application can be accessed. The modeless dialog is the opposite of a modal dialog. When a modeless dialog is active, the user is free to interact with the dialog and with the rest of the application.

Ways of Creating GUI Applications

There are two ways to write a GUI application:

From scratch using a simple text editor.

With Qt Designer, a visual design tool that comes with PyQt.

Obviously, you will be using Qt Designer for developing GUI applications in PyQt. Before you do that, to understand the structure of a GUI application, let's create one through coding.

Creating a GUI Application with Code

The application that you are going to create will display a pushbutton with the text Close on it. When you click the Close button, the application will terminate. Type the code below in any text editor and save the file with the extension `.pyw`. However, don't include the line numbers in the code, as they are just meant to identify each statement individually to explain their role.

Note

The console applications that you created before this chapter were saved with the `.py` extension. The GUI applications that you are going to develop now will be saved with the *.pyw* extension. This is to invoke the `Pythonw.exe` interpreter instead of the `Python.exe` interpreter so that no console window appears on executing a Python GUI application.

```
1. import sys
2. from PyQt4 import QtGui, QtCore
3. class demowind(QtGui.QWidget):
4.   def __init__(self, parent=None):
```

```
5.      QtGui.QWidget.__init__(self, parent)
6.      self.setGeometry(300, 300, 200, 200)
7.      self.setWindowTitle('Demo window')
8.      quit = QtGui.QPushButton('Close', self)
9.      quit.setGeometry(10, 10, 70, 40)
10.      self.connect(quit, QtCore.SIGNAL('clicked()'), QtGui.qApp,
          QtCore.SLOT('quit()'))
11. app = QtGui.QApplication(sys.argv)
12. dw = demowind()
13. dw.show()
14. sys.exit(app.exec_())
```

Before running this application, let's see what the code in different lines does.

1, 2. Imports the necessary modules. The basic GUI widgets are located in the QtGui module.

3. `QWidget` is the base class of all user interface objects in PyQt4, so you are creating a new `demowind` class that inherits from the base class, `QWidget`.

4, 5. Provides the default constructor for `QWidget`. The default constructor has no parent, and a widget with no parent is known as a window.

6. `setGeometry()` sets the size of the window and defines where to place it. The first two parameters are the x and y locations at which the window will be placed. The third is the width, and the fourth is the height of the window. A window 200 pixels high and wide will be positioned at coordinates 300,300.

7. This statement sets the window title to Demo Window. The title will be visible in the title bar.

8. Creates a pushbutton with the text Close.

9. Defines the width and height of the pushbutton as 70 and 40 pixels, respectively, and positioning it on the `QWidget` (window) at coordinates 10,10.

10. Event handling in PyQt4 uses signals and slots. A signal is an event, and a slot is a method that is executed on occurrence of a signal. For example, when you click a pushbutton, a `clicked()` event, also known as a signal, occurs, or is said to be *emitted*. The `QtCore.QObject.connect()` method connects signals with slots. In this case, the slot is a predefined PyQt4 method: `quit()`. That is, when the user clicks the pushbutton, the `quit()` method will be invoked. You will learn about event handling in detail soon.

11. Creates an application object with the name `app` through the `QApplication()` method of the QtGui module. Every PyQt4 application must create an

application object. `sys.argv`, which contains a list of arguments from the command line, is passed to the method while creating the application object. `sys.argv` helps in passing and controlling the startup attributes of a script.

12. An instance of the `demowind` class is created with the name `dw`.
13. The `show()` method will display the widget on the screen.
14. Begins the event handling loop for the application. The event handling loop waits for an event to occur and then dispatches it to perform some task. The event-handling loop continues to work until either the `exit()` method is called or the main widget is destroyed. The `sys.exit()` method ensures a clean exit, releasing memory resources.

Note

The `exec_()` method has an underscore because `exec` is a Python keyword.

On executing the above program, you get a window titled Demo Window with a pushbutton with text Close on it, as shown in Figure 7.4. When the pushbutton is selected, the `quit()` method will be executed, terminating the application.

Now, let's see how the Qt Designer tool, which comes with PyQt, makes the task of creating user interfaces quicker and easier.

Using Qt Designer

Though you can write PyQt programs from scratch using a simple text editor, you can also use Qt Designer, which comes with PyQt. For developing GUI applications in PyQt, using Qt Designer is a quick and easy way to design user interfaces without writing a single line of code. To open Qt Designer, click the Start button and then select All Programs > PyQt GPL v4.8.5 for Python v3.2 (x86) > Qt Designer.

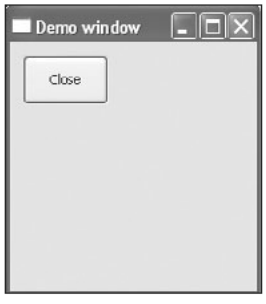


Figure 7.4
Output displaying the Close pushbutton.

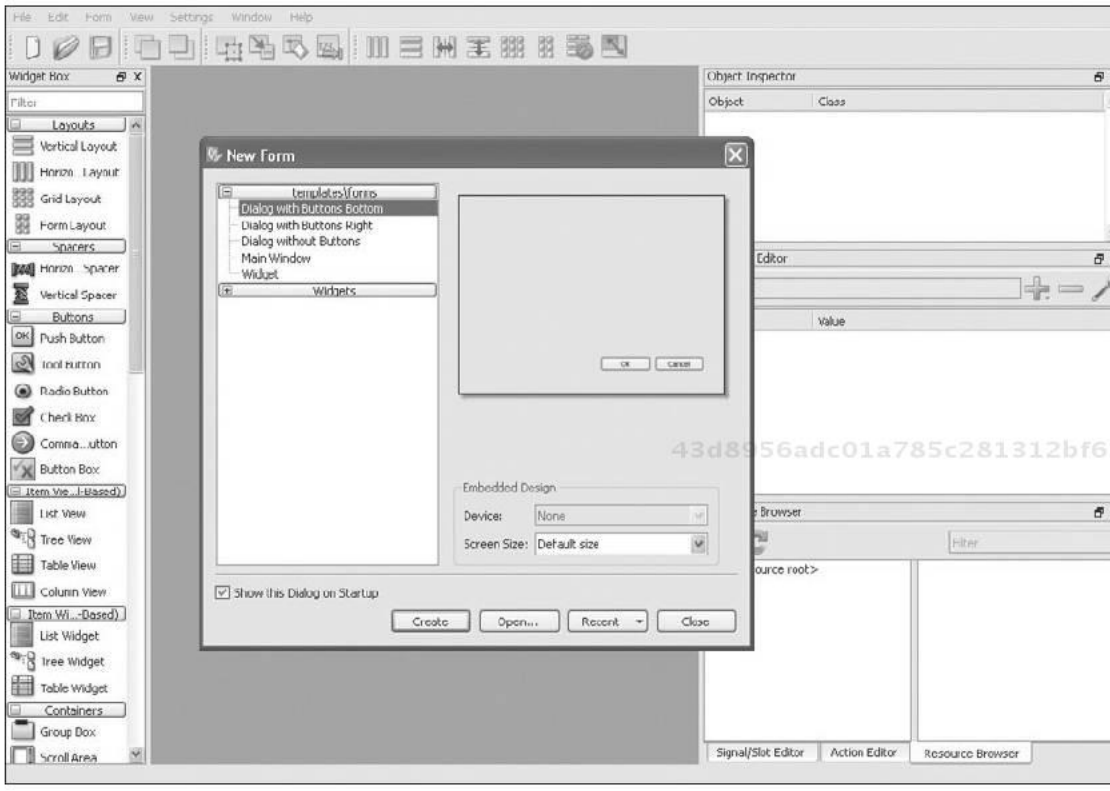


Figure 7.5
First screen on opening Qt Designer.

Qt Designer is for building graphical user interfaces. It makes it very easy for you to create dialogs or main windows using predefined templates, as shown in Figure 7.5. Qt Designer provides predefined templates for a new application:

- **Dialog with buttons at the bottom:** Creates a form with OK and Cancel buttons in the right bottom corner.
- **Dialog with buttons on the right:** Creates a form with OK and Cancel buttons on the right side.
- **Dialog without buttons:** Creates an empty form on which you can place widgets. The superclass for dialogs is `QDialog`. You will learn more about these classes soon.
- **Main window:** Provides a main application window with a menu bar and a toolbar that can be removed if not required.
- **Widget:** Creates a form whose superclass is `QWidget` rather than `QDialog`.

Note

When creating a GUI application, you need to specify a top-level widget, which is usually `QDialog`, `Qwidget`, or `QMainWindow`. If you create an application based on the Dialog template, the top-level widget or the first class that you inherit is `QDialog`. Similarly, if the application is based on the Main Window template, the top-level widget will be `QmainWindow`, and if you use the Widget template for your application, the top-level widget will be `QWidget`. The widgets that you use for the user interface are then treated as child widgets of the classes.

Qt Designer displays a menu bar and toolbar at the top. It shows a Widget Box on its left that contains a variety of widgets used to develop applications, grouped in sections. All you have to do is drag and drop the widgets you want from the form. You can arrange widgets in layouts, set their appearance, provide initial attributes, and connect their signals to slots. The user interface that you create with Qt Designer is stored in a `.ui` file that includes all the form's information: its widgets, layout, and so on. The `.ui` file is an XML file, and you need to convert it to Python code. That way, you can maintain a clear separation between the visual interface and the behavior implemented in code. You will soon see the methods of converting `.ui` files into Python code.

Note

You can create widgets with code, also.

On the right side of Qt Designer you will find three windows by default, as shown in Figure 7.6.

- **Object Inspector:** Displays a hierarchical list of all the objects present on the form. You can select any object on a form by clicking on its corresponding name in the Object Inspector. Usually you select an object in Object Inspector window when you have overlapping objects. The window also displays the layout state of the containers. Containers are those widgets that can store other widgets or objects. Containers include frames, group boxes, stacked widgets, tab widgets, and tool box widgets.
- **Property Editor:** Used to view and change the properties of the form and widgets. It consists of two columns, Property and Value. The Property column lists property names, and the Value column lists the corresponding values. To change a property to the same value for a set of widgets, select all of them. To select a set of widgets, click one of the widgets and then Shift+Click the others one by one. When a set of widgets is selected, the Property Editor window will show the

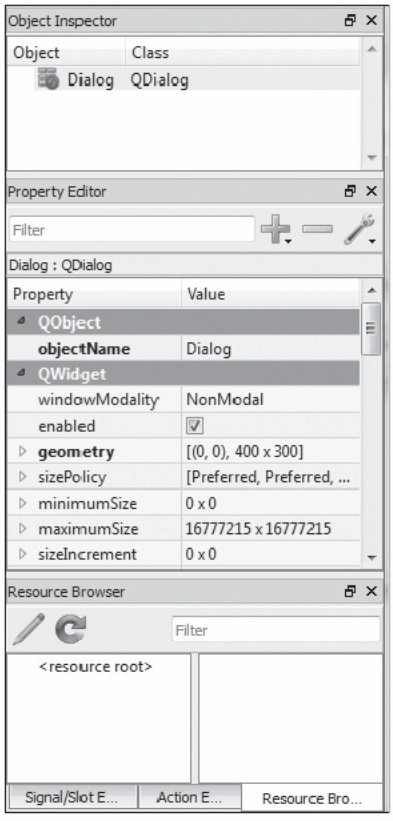


Figure 7.6
Three windows: Object Inspector, Property Editor, and Resource Browser.

properties that are common in all the selected widgets, and any change made to one property will be applied to the selected widgets.

- **Resource Browser**: Qt Designer enables you to maintain resources like images, audio, video, etc., of your applications through the Resource Browser. For each form of your application, a separate resource file is maintained. You can define, load, and edit resource files of your application through the Resource Browser. Below the Resource Browser window, you find two more tabs, the Signal/Slot Editor and Action Editor.
- **Signal/Slot Editor:** This window displays the signal/slot connections between objects. You can edit the signal/slot connections through this window.
- **Action Editor:** The Action Editor lets you to manage the actions of your applications. To initiate actions, the toolbar and menu bar are designed in an application. The respective action or task for each of the icons of the toolbar and menu items of the menu bar are defined through the Action Editor. You can

create new actions, delete actions, edit actions, and define icons for the actions through the Action Editor. Also, you can associate respective actions with menu items and toolbars.

Note

For quick and handy actions, Qt Designer provides a context menu that you get by right-clicking an object.

The main component used for creating a user interface is widgets. Button, menus, and scrollbars are examples of widgets and are not only used for receiving user input but also for displaying data and status information. Widgets can be nested inside another in a parent-child relationship. A widget that has no parent widget is called a window. The class for widgets, `QWidget`, provides methods to render them on screen, receive user input, and handle different events. All UI elements that Qt provides are subclasses of `QWidget`. Qt Designer displays a list of widgets in a Widget Box displayed on the left side.

Widget Box

The Widget Box (see Figure 7.7) displays a categorized list of widgets and other objects that you can use for designing a user interface quickly and easily. Widgets with similar functions and uses are grouped into categories. It's very simple to create a graphical user interface by switching to Widget Editing mode. Select an icon from the toolbar and drag the desired widgets to the form.

You can also group widgets that you use often in a category you create, also known as a scratch pad category. To place widgets in the scratch pad category, simply drag them from the form and drop them into the category. These widgets can be used in the same way as any other widget. You can change the name of any widget and remove it from the scratch pad with the context menu.

Widgets are objects of their respective classes. (Qt Designer does not use class names for its widgets; the name of the widget signifies the class it refers to.)

The widgets in Widget Box are grouped into the following categories:

- Layouts
- Spacers
- Buttons
- Item Views (Model-Based)

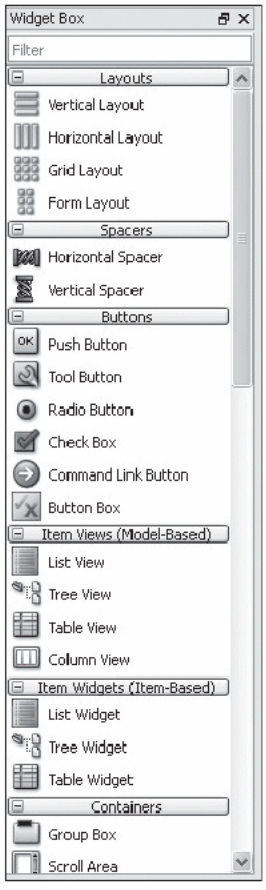


Figure 7.7
Widget Box.

- Item Widgets (Item-Based)
- Containers
- Input Widgets
- Display Widgets
- Phonon

A description of the widgets in each category is as follows.

Layouts

Layouts are used for arranging widgets in a desired manner. The layout controls the size of the widgets within it, and widgets are automatically resized when the form is resized. The widgets in the Layouts group is shown in Table 7.1.

Table 7.1 Widgets in the Layouts Group

Widget	Description
Vertical Layout (`QVBoxLayout`)	Arranges widgets vertically, one below the other.
Horizontal Layout (`QHBoxLayout`)	Arranges widgets horizontally, one next to the other.
Grid Layout (`QGridLayout`)	Arranges widgets into rows and columns.
Form Layout (`QFormLayout`)	Arranges widgets in a two-column layout. The first column usually displays message(s) in labels, and the second column usually contains the widgets, enabling the user to enter/edit data corresponding to the labels in the first column.

Table 7.2 Widgets in the Spacers Group

Widget	Description
Horizontal Spacer (Spacer)	Inserts horizontal spaces between widgets.
Vertical Spacer (Spacer)	Inserts vertical spaces between widgets.

Spacers

Spacers are not visible while running a form and are used for inserting spaces between widgets or groups of widgets. The widgets in the Spacers group are shown in Table 7.2.

Buttons

Buttons are used to initiate an action. They are event or signal generators that can be used to perform tasks. The widgets in the Buttons group are shown in Table 7.3.

Item Views (Model-Based)

Item Views widgets are used for displaying large volumes of data. Model-based means that the widgets are part of a model/view framework and enable you to present data in different formats and through multiple views. The classes of these widgets implement the interfaces defined by the `QAbstractItemView` class to allow it to

Table 7.3 Widgets in the Buttons Group

Widget	Description
Push Button (`QPushButton`)	Displays a command button.
Tool Button (`QToolButton`)	Displays a button to access commands or options. Used inside a toolbar.
Radio Button (`QRadioButton`)	Displays a radio button with a text label.
Check Box (`QCheckBox`)	Displays a check box with a text label.
Command Link Button (`QCommandLinkButton`)	Displays a command link button.
Button Box (`QDialogButtonBox`)	A sub-class of `QWidget` that presents a set of buttons in a layout.

Table 7.4 Widgets in the Item Views (Model-Based) Group

Widget	Description
List View (`QListView`)	Used to display a list of items. Must be used with a `QAbstractItemModel` subclass.
Tree View (`QTreeView`)	Used to display hierarchical data. Must be used with a `QAbstractItemModel` subclass.
Table View (`QTableView`)	Used to display data in tabular form. Can display icons as well as text in every cell. Must be used in conjunction with a `QAbstractItemModel` subclass.
Column View (`QColumnView`)	Provides a model/view implementation of a column view. It displays data in a number of list views.

display data provided by models derived from the `QAbstractItemModel` class. The widgets in the Item Views (Model-Based) group are shown in Table 7.4.

Item Widgets (Item-Based)

Item Widgets have self-contained views. The widgets in the Item Widgets (Item-Based) group are shown in Table 7.5.

Table 7.5 Widgets in the Item Widgets (Item-Based) Group

Widget	Description
List Widget (`QListWidget`)	Used to display a list of items. It has a built-in model, so items can be added to it directly.
Tree Widget (`QTreeWidget`)	Used to display hierarchical data. It has a built-in model, so items can be added to it directly.
Table Widget (`QTableWidget`)	Used to display data in tabular form. Can display icons as well as text in every cell. It has a built-in model, so items can be added to it directly.

Table 7.6 Widgets in the Containers Group

Widget	Description
Group Box (`QGroupBox`)	Used to group together a collection of widgets of similar function.
Scroll Area (`QScrollArea`)	Used to display the contents of a child widget within a frame. If the child widget exceeds the size of the frame, scrollbars appear to enable you to view the entire child widget.
Tool Box (`QToolBox`)	Displays a series of pages or sections in a tool box.
Tab Widget (`QTabWidget`)	Displays tabs that can be used to display information. A large volume of information can be displayed by splitting it into chunks and displaying it under individual tabs
Stacked Widget (`QStackedWidget`)	Displays a stack of widgets where only one widget is visible at a time.
Frame (`QFrame`)	Used to enclose and group widgets. Can also be used as a placeholder in forms.
Widget (`QWidget`)	The base class of all user interface objects.
MdiArea (`QMdiArea`)	Provides an area for displaying MDI windows.
Dock Widget (`QDockWidget`)	Can be docked inside a main window or floated as an independent tool window.

Containers

Container widgets are used to control a collection of objects on a form. A widget dropped onto a container becomes a child object of the container. The child objects in a container can also be arranged in desired layouts. The widgets in the Containers group are shown in Table 7.6.

Table 7.7 Widgets in the Input Widgets Group

Widget	Description
Combo Box (`QComboBox`)	Displays a pop-up list.
Font Combo Box (`QFontComboBox`)	Displays a combo box that allows font selection.
Line Edit (`QLineEdit`)	Displays a single-line text box for entering/editing plain text.
Text Edit (`QTextEdit`)	Used to edit plain text or HTML.
Plain Text Edit (`QPlainTextEdit`)	Used to edit and display plain text.
Spin Box (`QSpinBox`)	Displays a spin box.
Double Spin Box (`QDoubleSpinBox`)	Displays a spin box for double values.
Time Edit (`QTimeEdit`)	Used for editing times.
Date Edit (`QDateEdit`)	Used for editing dates.
Date/Time Edit (`QDateTimeEdit`)	Used for editing dates and times.
Dial (`QDial`)	Displays a rounded range control.
Horizontal Scrollbar (`QScrollBar`)	Displays a horizontal scrollbar.
Vertical Scrollbar (`QScrollBar`)	Displays a vertical scrollbar.
Horizontal Slider (`QSlider`)	Displays a horizontal slider.
Vertical Slider (`QSlider`)	Displays a vertical slider.
`QsciScintilla`	Scintilla is an editing component that performs syntax styling, code completion, break points, auto indenting, and other tasks. It is very useful in editing and debugging source code. The Scintilla component is inside `QsciScintilla` and used in Qt Designer for developing GUI applications like any other Qt widget.

Input Widgets

Input Widgets are for used for interacting with the user. The user can supply data to the application through these widgets. The widgets in the Input Widgets group are shown in Table 7.7.

Display Widgets

Display widgets are used for displaying information or messages to the user. The widgets in the Display Widgets group are shown in Table 7.8.

Table 7.8 Widgets in the Display Widgets Group

Widget	Description
Label (`QLabel`)	Displays text or images.
Text Browser (`QTextBrowser`)	Displays a read-only multiline text box that can display both plain text and HTML, including lists, tables, and images. It supports clickable links as well as cascading style sheets.
Graphics View (`QGraphicsView`)	Used to displays graphics.
Calendar (`QCalenderWidget`)	Displays a monthly calendar allowing you to select a date.
LCD Number (`QLCDNumber`)	Displays digits in LCD-like display.
Progress Bar (`QProgressBar`)	Displays horizontal and vertical progress bars.
Horizontal Line (`QFrame`)	Displays a horizontal line.
Vertical Line (`QFrame`)	Displays a vertical line.
`QDeclarativeView`	A `QGraphicsView` subclass provided for displaying QML interfaces. To display a QML interface within QWidget-based GUI applications that do not use the Graphics View framework, `QDeclarative` is used. `QDeclarativeView` initializes `QGraphicsView` for optimal performance with QML so that user interface objects can be placed on a standard `QGraphicsScene` and displayed with `QGraphics-View`. QML is a declarative language used to describe the user interface in a tree of objects with properties.
`QWebView`	Used to view and edit web documents.

Phonon

Phonon is a multimedia API that provides an abstraction layer for capturing, mixing, processing, and playing audio and video. The widgets in the Phonon group are shown in Table 7.9.

Qt Designer displays a toolbar at the top that shows icons for frequently used tasks such as opening and saving files, switching modes, and applying layouts. Let's look at the toolbar.

Table 7.9 Widgets in the Phonon Group

Widget	Description
`Phonon::VideoPlayer`	Used to display video.
`Phonon::SeekSlider`	Displays slider for setting positions in media stream.
`Phonon::VolumeSlider`	Displays slider to control volume of audio output.

Toolbar

At the top of Qt Designer is a toolbar with icons as shown in Figure 7.8.

The following is a brief description of icons shown in the toolbar:

- **New:** Displays a New Form dialog box (refer to Figure 7.5) showing different templates for creating a new form.
- **Open:** Opens the Open Form dialog box, which you can use to browse your disk drive to search and select a `.ui` file to work on.
- **Save:** Used to save the form.
- **Send to Back:** Sends the selected widget to the back in overlapping widgets, making it invisible. Consider two overlapping pushbuttons, `PushButton1` and `PushButton2`. If you select `PushButton1` and click Send to Back, `PushButton1` will be hidden behind `PushButton2`, as shown in Figure 7.9 (a).
- **Bring to Front:** Brings the selected widget to the front, making it visible. This icon works only when widgets overlap each other. If you select `PushButton1` and click Bring to Front, it will become visible, as shown in Figure 7.9(b).

Figure 7.8
Qt Designer toolbar.

Figure 7.9
(a) `PushButton1` sent back. (b) `PushButton1` brought to the front.

- **Edit Widgets:** The Widget Editing mode allows you to edit widget properties. Also, you can drop widgets into existing layouts on the form in this mode. You can also drag widgets between forms. You can also clone a widget by dragging it with the Ctrl key pressed. To activate the Widget Editing mode, you can choose any of the three options: press F3, select the Edit > Edit Widgets from the menu, or click the Edit Widgets icon on the toolbar.
  - **Edit Signals/Slots:** The Signals and Slots Editing mode is used for representing and editing signal/slot connections between objects on a form. To switch to the Signals and Slots Editing mode, you can press the F4 key, select the Edit > Edit Signals/Slots option, or select the Edit Signals/Slots icon from the toolbar. The mode displays all the signal and slot connections in the form of arrows so that you know which object is connected to what. You can also create new signal and slot connections between widgets in this mode and delete an existing signal. The signals and slots refer to different events and corresponding methods that are executed on occurrence of an event. To establish signal and slot connection between two objects in a form, select an object by clicking with the left mouse button and drag the mouse towards the object to which you want to connect and release the mouse button. You can also cancel the connection while dragging the mouse by pressing the Esc key. When you release the mouse over the destination object, a Connection Dialog box appears, prompting you to select a signal from the source object and a slot from the destination object. After selecting the respective signal and slot, select OK to establish the signal/slot connection. You can also select Cancel in the Connection dialog box to cancel the connection operation. The selected signal and slot will appear as labels in the arrow connecting the two objects. To modify a connection, double-click the connection path or one of its labels to display the Connection dialog box. From the Connection dialog you can edit a signal or a slot as desired. To delete a signal/slot connection, select its arrow on the form and press the Del key. The signal/slot connection can also be established between an object and the form; you can connect signals from objects to the slots in the form. To connect an object to the form, select the object, drag the mouse, and release the mouse button over the form. The end point of the connection changes to the electrical ground symbol. To come out of the Signals and Slots Editing mode, select Edit > Edit Widgets or press the F3 key.
  - **Edit Buddies:** Buddy Editing mode is used for setting keyboard focus on the widgets that cannot accept keyboard input. That is, by making a widget that can accept keyboard input a buddy, widgets that cannot accept keyboard input will also gain keyboard focus. Arrows appear to show the relationships between

widgets and their buddies. To activate the Buddy Editing mode, you can either select the Edit > Edit Buddies option from the menu, or click the Edit Buddies icon on the toolbar.

- **Edit Tab Order:** In this mode, you can specify the order in which input widgets can get keyboard focus. The default tab order is based on the order in which widgets are placed on the form.
- **Lay Out Horizontally:** Arranges selected widgets in a horizontal layout next to each other. Shortcut key is Ctrl+1.
- **Lay Out Vertically:** Arranges selected widgets in a vertical layout, one below another. Shortcut key is Ctrl+2.
- **Lay Out Horizontally in Splitter:** In this layout, the widgets are placed in a splitter, arranged horizontally and allowing you to adjust the amount of space allocated to each widget. Shortcut key is Ctrl+3.
- **Lay Out Vertically in Splitter:** The widgets are arranged vertically, allowing the user to adjust the amount of space allocated to each widget. Shortcut key is Ctrl+4.
- **Lay Out in a Grid:** Arranges widgets in a table-like grid (rows and columns). Each widget occupies one table cell that you can modify to span several cells.
- **Lay Out in a Form Layout:** Arranges selected widgets in a two-column format. The left column is usually for Label widgets displaying messages, and the right column shows widgets for entering/editing/showing data for the corresponding labels in the first column, such as Line Edit, Combo Box, and Date Edit.
- **Break Layout:** Once widgets are arranged in a layout, you cannot move and resize them individually, as their geometry is controlled by the layout. This icon is to break the layout. Shortcut key is Ctrl+0.
- **Adjust Size:** Adjusts the size of the layout to accommodate contained widgets and to ensure that each has enough space to be visible. Shortcut key is Ctrl+J.

In almost all applications, you need some very fundamental widgets such as Labels, Line Edits, and Push Buttons. These widgets are required to display text messages, to accept input from the user, and to initiate some action, respectively. Let's look at these fundamental widgets.

Understanding Fundamental Widgets

The first widget we will discuss is the Label widget, a very popular way of displaying text or information in a GUI application.

Table 7.10 Methods Provided by the QLabel Class

Methods	Usage
`setText()`	Assigns text to the Label widget.
`setPixmap()`	Assigns a pixmap, an instance of the `QPixmap` class, to the Label widget.
`setNum()`	Assigns an integer or double value to the Label widget.
`clear()`	Clears text from the Label widget.

Displaying Text

To display non-editable text or an image, Label widgets are used; a Label is an instance of the `QLabel` class. A Label widget is a very popular widget for displaying messages or information to the user. The methods provided by the `QLabel` class are shown in Table 7.10.

The default text of a `QLabel` is TextLabel. That is, when you add a `QLabel` to a form by dragging a Label widget and dropping it on the form, it will display "TextLabel." Besides using `setText()`, you can also assign text to a selected `QLabel` by setting its `text` property in the Property Editor window. For instance, if you set the `text` property to **Enter your name**, the selected `QLabel` will show the text "Enter your name" on the form.

You can also set any letter in the `QLabel` text to act as a shortcut key by preceding it with an ampersand symbol (&). For instance, if you set the `text` property of the selected `QLabel` to **&Enter your name**, the letter E will become a shortcut key, and you can access that `QLabel` with the Alt+E keys.

Entering Single-Line Data

To allow the user to enter or edit single-line data, you use the Line Edit widget, which is an instance of `QLineEdit`. The widget supports simple editing mechanisms such as undo, redo, cut, and paste. The methods provided by `QLineEdit` are shown in Table 7.11.

Signals emitted by the Line Edit widget are these:

`textChanged()`: The signal is emitted when text in the Line Edit widget is changed.

`returnPressed()`: The signal is emitted when Return or Enter is pressed.

Table 7.11 Methods Provided by QLineEdit

Method	Usage
`setEchoMode()`	Used to set the echo mode of the Line Edit widget to determine how the contents of the Line Edit widget are displayed. The available options are these: `Normal`: Default mode. Displays characters as they are entered. `NoEcho`: Doesn't display anything. `Password`: Displays asterisks as the user enters data. `PasswordEchoOnEdit`: Displays characters when editing; otherwise, asterisks are displayed.
`maxLength()`	Used to specify the maximum length of text that user can enter. For multiline editing, you use `QTextEdit`.
`setText()`	Assigns text to the Line Edit widget.
`text()`	Fetches the text entered in the Line Edit widget.
`clear()`	Clears the contents of the Line Edit widget.
`setReadOnly()`	Passes the Boolean value true to this method to make the Line Edit widget read-only. The user cannot edit the contents of the Line Edit widget but can copy it. The cursor will become invisible in read-only mode.
`isReadOnly()`	Returns true if the Line Edit widget is in read-only mode.
`setEnabled()`	The Line Edit widget will be blurred, indicating that it is disabled. You cannot edit content in a disabled Line Edit widget, but you can assign text via the `setText()` method.
`setFocus()`	Used to set the cursor on the specified Line Edit widget.

`editingFinished()`: The signal is emitted when focus is lost on the Line Edit widget, confirming the editing task is over on it.

The next widget is the most common way of initiating actions in any application.

Displaying Buttons

To display pushbuttons (usually command buttons) in an application, you need to create an instance of the `QPushButton` class. When assigning text to buttons, you can create shortcut keys by preceding any character in the text with an ampersand. For example, if the text assigned to a pushbutton is `&Click Me`, the character `C` will be underlined to indicate that it is a shortcut key, and the user can select the button

by pressing Alt+C. The button emits a `clicked()` signal if it is activated. Besides text, an icon can also be displayed in the pushbutton. The methods for displaying text and an icon in a pushbutton are these:

`setText()`: Used to assign the text to the pushbutton.

`setIcon()`: Used to assign icon to the pushbutton.

The only concept left to examine before you begin with your first application in Qt Designer is event handling. Let's see how events are handled in PyQt.

Event Handling in PyQt

In PyQt, the event-handling mechanism is also known as *signals and slots*. Every widget emits signals when its state changes. Whenever a signal is emitted, it is simply thrown. To perform a task in response to a signal, the signal has to be connected to a slot. A *slot* refers to the method containing the code that you want to be executed on occurrence of a signal. Most widgets have predefined slots, you don't have to write code for connecting a predefined signal to a predefined slot. To respond to the signals emitted, you identify the `QObject` and the signal it emits and invoke the associated method. You can use Qt Designer for connecting signals with built-in slots. How? Let's see by creating an application.

Note

Signals differ according to the widget type.

First Application in Qt Designer

Let's create an application in Qt Designer to demonstrate how to connect signals with built-in slots. On opening, Qt Designer asks you to select a template for your new application, as shown previously in Figure 7.5. Qt Designer provides a number of templates that are suitable for different kinds of applications. You can choose any of these templates and then select the Create button. Select Dialog with Buttons Bottom and click the Create button. A new form will be created with an "untitled" caption. The form contains a button box that has two buttons, OK and Cancel, as shown in Figure 7.10. The signal-slot connections of the OK and Cancel buttons are already set up by default.

In order to learn how to connect signals with slots manually, select the button box by clicking either of the buttons, and then delete it (which removes the buttons). Now you have an entirely blank form. Add a `QLabel`, `QLineEdit`, and `QPushButton` to

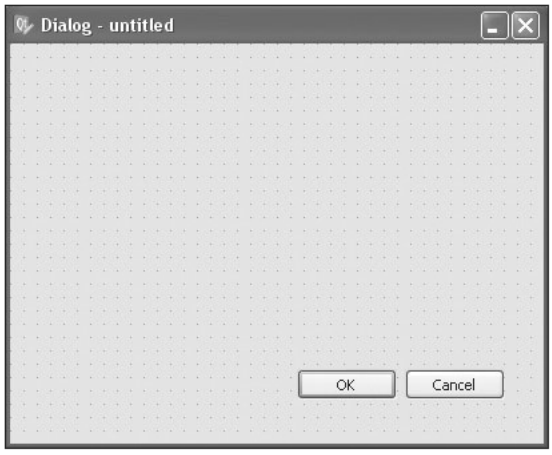


Figure 7.10
Dialog box with two buttons, OK and Cancel.

the form by dragging and dropping a Label, Line Edit, and Push Button widget from the Widget Box on the form. The default `text` property of Label is `TextLabel`, as shown in Figure 7.11(a). You can change it by changing the `text` property in the Property Editor. Select the Label widget and set its `text` property to Enter Text through the Property Editor. Similarly, set the text of the Push Button widget to Clear, as shown in Figure 7.11(b).

Note

To preview a form while editing, select either Form, Preview or Ctrl+R.

You want some action to happen when the user selects Clear on the form, so you need to connect Push Button's signal to Line Edit's slot.

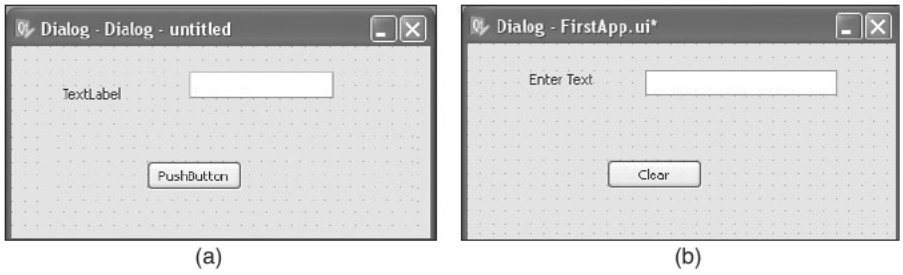


Figure 7.11
(a) Three widgets dropped on the form. (b) Widgets on the form with the `text` property set.

Connecting to Predefined Slots

Currently, you are in widget editing mode, and to apply signal/slot connections, you need to first switch to signals and slots editing mode. Select the Edit Signals/Slots icon from the toolbar to switch to signals and slots editing mode. On the form, select the Clear button and drag the mouse to the Line Edit widget and release the mouse button. The Configure Connection dialog will pop up, allowing you to establish a signal-slot connection between the Clear button and the Line Edit widget, as shown in Figure 7.12.

When the user selects the Clear button, you want any text in the Line Edit widget to be deleted. For this to happen, you have to connect the pushbutton's `clicked()` signal to the Line Edit's `clear()` slot. So, in the Configure Connection dialog, select the `clicked()` signal from the Push Button column and the `clear()` slot from the Line Edit column and select OK. On the form, you will see that an arrow appears, representing the signal-slot connection between the two widgets as shown in Figure 7.13.

Let's save the form with the name FirstApp. The default location where the form will be saved is C:\Python32\Lib\site-packages\PyQt4. The form will be saved in a file with the `.ui` extension. The `FirstApp.ui` file will contain all the information of

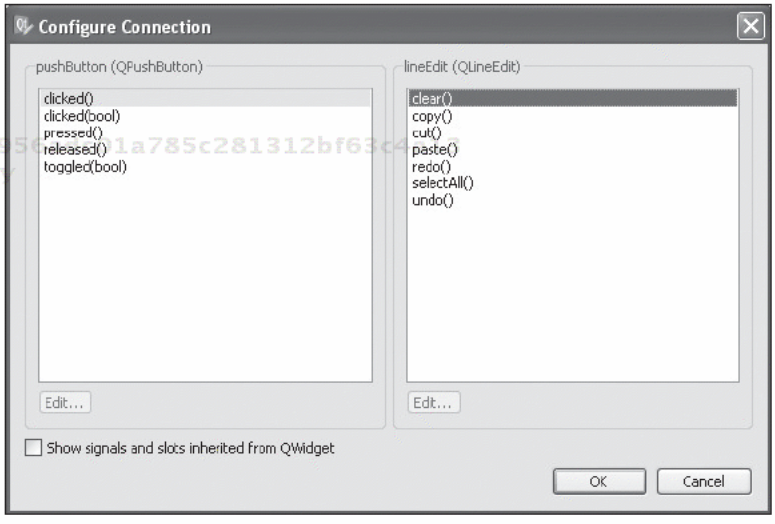


Figure 7.12
Configure Connection dialog displaying predefined slots.

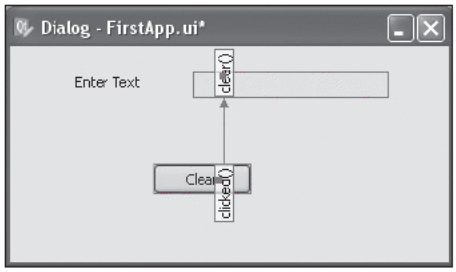


Figure 7.13
The signal-slot connection in widgets represented with arrows.

the form, its widgets, layout, and so on. The `.ui` file is an XML file, and it contains the following code:


```
FirstApp.ui
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>Dialog</class>
 <widget class="QDialog" name="Dialog">
  <property name="geometry">
  <rect>
   <x>0</x>
   <y>0</y>
   <width>337</width>
   <height>165</height>
  </rect>
 </property>
 <property name="windowTitle">
  <string>Dialog</string>
 </property>
 <widget class="QPushButton" name="pushButton">
  <property name="geometry">
   <rect>
    <x>110</x>
    <y>90</y>
    <width>75</width>
    <height>23</height>
   </rect>
  </property>
  <property name="text">
   <string>Clear</string>
  </property>
 </widget>
 <widget class="QLineEdit" name="lineEdit">
```

```
  <property name="geometry">
   <rect>
    <x>140</x>
    <y>20</y>
    <width>151</width>
    <height>20</height>
   </rect>
  </property>
 </widget>
 <widget class="QLabel" name="label">
  <property name="geometry">
   <rect>
    <x>50</x>
    <y>20</y>
    <width>71</width>
    <height>16</height>
   </rect>
  </property>
  <property name="text">
   <string>Enter Text</string>
  </property>
 </widget>
</widget>
<resources/>
<connections>
 <connection>
  <sender>pushButton</sender>
  <signal>clicked()</signal>
  <receiver>lineEdit</receiver>
  <slot>clear()</slot>
  <hints>
   <hint type="sourcelabel">
    <x>161</x>
    <y>103</y>
   </hint>
   <hint type="destinationlabel">
    <x>164</x>
    <y>24</y>
   </hint>
  </hints>
 </connection>
</connections>
</ui>
```

To use the file, you first need to convert it into Python script. The command utility that you will use for converting a `.ui` file into a Python script is `pyuic4`. In Windows, the `pyuic4` utility is bundled with PyQt. To do the conversion, you need to open a command prompt window and navigate to the folder where the file is saved and issue this command:

```
C:\Python32\Lib\site-packages\PyQt4>pyuic4 FirstApp.ui -o FirstApp.py
```

Recall that you saved the form at the default location, C:\Python32\Lib\site-packages\PyQt4.

The command shows the conversion of the `FirstApp.ui` file into a Python script, `FirstApp.py`.

Note

The Python code generated by this method should not be modified manually, as any changes will be overwritten the next time you run the `pyuic4` command.

The Python script file `FirstApp.py` may have the following code. Your generated code may slightly vary when compared with the following code, as it depends on several factors, including window size, button location, and so on:

```
FirstApp.py
# Form implementation generated from reading ui file 'FirstApp.ui'

from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_Dialog(object):
  def setupUi(self, Dialog):
    Dialog.setObjectName(_fromUtf8("Dialog"))
    Dialog.resize(337, 165)
    self.pushButton = QtGui.QPushButton(Dialog)
    self.pushButton.setGeometry(QtCore.QRect(110, 90, 75, 23))
    self.pushButton.setObjectName(_fromUtf8("pushButton"))
    self.lineEdit = QtGui.QLineEdit(Dialog)
    self.lineEdit.setGeometry(QtCore.QRect(140, 20, 151, 20))
    self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
    self.label = QtGui.QLabel(Dialog)
    self.label.setGeometry(QtCore.QRect(50, 20, 71, 16))
    self.label.setObjectName(_fromUtf8("label"))
```

```
        self.retranslateUi(Dialog)
        QtCore.QObject.connect(self.pushButton, QtCore.SIGNAL(_fromUtf8("clicked()")),
self.lineEdit.clear)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

    def retranslateUi(self, Dialog):
        Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None,
QtGui.QApplication.UnicodeUTF8))
        self.pushButton.setText(QtGui.QApplication.translate("Dialog", "Clear", None,
QtGui.QApplication.UnicodeUTF8))
        self.label.setText(QtGui.QApplication.translate("Dialog", "Enter Text", None,
QtGui.QApplication.UnicodeUTF8))
```

This script is very easy to understand. A class with the name of the top-level object is created, with `Ui_` prepended. Since, the top-level object used in our application is `Dialog`, the class `Ui_Dialog` is created and stores the interface elements of our widget. That class has two methods, `setupUi()` and `retranslateUi()`. The `setupUi()` method sets up the widgets; it creates the widgets that you used while defining the user interface in Qt Designer. The method creates the widgets one by one and also sets their properties. The `setupUi()` method takes a single argument, which is the top-level widget in which the user interface (child widgets) is created. In our application, it is an instance of `QDialog`. The `retranslateUi()` method translates the interface. The file imports everything from both modules, `QtCore` and the `QtGui`, as you will be needing them in developing GUI applications.

`QtCore`: The `QtCore` module forms the foundation of all Qt-based applications. It contains the most fundamental classes, such as `QCoreApplication`, `QObject`, and so on. These classes perform several important tasks, such as file handling, event handling through the event loop, implementing the signals and slot mechanism, concurrency control, and much more. The module includes several classes, including `QFile`, `QDir`, `QIODevice`, `QTimer`, `QString`, `QDate`, and `QTime`.

`QtGui`: The `QtGUI` module contains the classes required in developing cross-platform GUI applications. The module contains the majority of the GUI classes, including `QCheckBox`, `QComboBox`, `QDateTimeEdit`, `QLineEdit`, `QPushButton`, `QPainter`, `QPaintDevice`, `QApplication`, `QTextEdit`, and `QTextDocument`.

You will be treating the code as a header file, and you will import it to the source file from which you will invoke its user interface design. Let's create the source file with the name `callFirstApp.pyw` and import the `FirstApp.py` code to it. The code in the file is as shown here:

```
callFirstApp.pyw
import sys
```

```
from FirstApp import *

class MyForm(QtGui.QDialog):
   def __init__(self, parent=None):
      QtGui.QWidget.__init__(self, parent)
      self.ui = Ui_Dialog()
      self.ui.setupUi(self)

if __name__ == "__main__":
   app = QtGui.QApplication(sys.argv)
   myapp = MyForm()
   myapp.show()
   sys.exit(app.exec_())
```

The `sys` module is imported to enable you to access the command-line arguments stored in the `sys.argv` list. First you create an `QApplication` object. Every PyQt GUI application must have a `QApplication` object to provide access to information such as the application's directory, screen size, and so on. When creating an `QApplication` object, you pass the command-line arguments to it for the simple reason that PyQt can act on command-line arguments if required. You create an instance of `MyForm` and call its `show()` method, which adds a new event to the `QApplication` object's event queue: a request to paint the widgets specified in the class, `MyForm`. The method `app.exec_()` is called to start the `QApplication` object's event loop. Once the event loop begins, the top-level widget used in the class, `MyForm`, is displayed along with its child widgets. All the events that occur, whether through user interaction or system-generated, are added to the event queue. The application's event loop continuously checks to see if any event has occurred. If so, the event loop processes it and eventually passes it to the associated method. When you close the top-level widget being displayed, it goes into hidden mode, and PyQt deletes the widget and performs a clean termination of the application.

In PyQt, any widget can be used as a top-level window. To declare `QDialog` as a top-level window, all you need is to declare the parent of the class `MyForm` as `None`. So to the `__init__()` method of our `MyForm` class, you pass a default parent of `None` to indicate that the `QDialog` displayed through this class is a top-level window.

Note

A widget that has no parent becomes a top-level window.

Recall that the user interface design is instantiated by calling the `setupUI()` method of the class that was created in the Python code (`Ui_Dialog`). What you need is to create an instance of the class `Ui_Dialog`, the class that was created in the Python

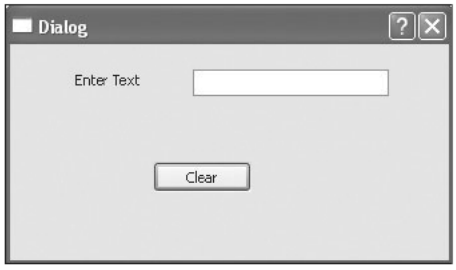


Figure 7.14
Output of FirstApp application.

code, and invoke its `setupUi()` method. The Dialog widget will be created as the parent of all the user interface widgets and displayed on the screen.

Note

`QDialog`, `QMainWindow`, and all PyQt's widgets are derived from `QWidget`.

On running the above Python script, the application prompts for text to be entered in the Line Edit widget, as shown in Figure 7.14.

Any text in the Line Edit widget will be deleted when you select the Clear button.

Congratulations on successfully creating and executing your first GUI application.

In this application, you saw how to connect the built-in signals with slots. What if you want a custom method to execute an occurrence of an event?

Using Custom Slots

The application you are going to create now will prompt a user to enter a name and select a pushbutton. When the pushbutton is selected, the application will display a welcome message to the user. This time let's use the Dialog without Buttons template, which provides a blank form ready to receive widgets. Recall that an instance of `QDialog` is the top-level widget in applications based on the Dialog template. Let's add two `QLabels`, a `QlineEdit`, and a `QPushButton` to the form by dragging and dropping two Label, Line Edit, and Push Button widgets from the Widget Box, as shown in Figure 7.15(a). Set the `objectName` property of the first and second Label to `labelEnterName` and `labelMessage`, respectively. Also, set the `objectName` property of the Line Edit and Push Button widgets to `lineUserName` and `ClickMeButton`, respectively. Set the `text` property of the first Label widget to **Enter your name**. Also, delete the default `text` property, `TextLabel`, from the

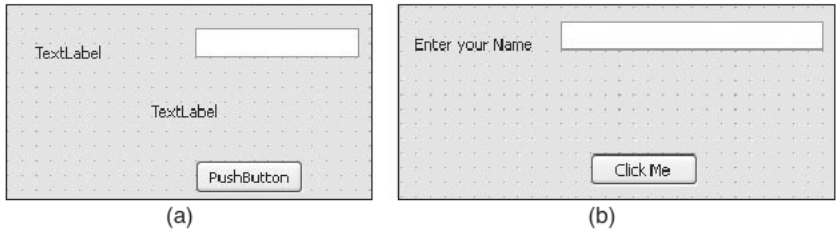


Figure 7.15
(a) Four widgets dropped on the form. (b) Widgets on the form with the `text` property set.

second Label as you will be setting its text through a Python script to display the welcome message to the user. The second Label will become invisible on deleting its `text` property. Also, set the text of the Push Button widget to **Click Me**, as shown in Figure 7.15(b).

Note

The `objectName` property helps in distinguishing widgets in the form, and it is only through the object names that the widgets are accessed in coding.

Save the form with the name `welcomemsg.ui`. You know that a `.ui` file is an XML file and has to be converted into Python code through the `pyuic4` command-line utility. The generated Python code is shown here:

```
welcomemsg.py
# Form implementation generated from reading ui file 'welcomemsg.ui'

from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_Dialog(object):
    def setupUi(self, Dialog):
        Dialog.setObjectName(_fromUtf8("Dialog"))
        Dialog.resize(400, 300)
        self.ClickMeButton = QtGui.QPushButton(Dialog)
        self.ClickMeButton.setGeometry(QtCore.QRect(150, 120, 75, 23))
        self.ClickMeButton.setObjectName(_fromUtf8("ClickMeButton"))
        self.labelEnterName = QtGui.QLabel(Dialog)
        self.labelEnterName.setGeometry(QtCore.QRect(30, 30, 101, 21))
        self.labelEnterName.setObjectName(_fromUtf8("labelEnterName"))
        self.labelMessage = QtGui.QLabel(Dialog)
        self.labelMessage.setGeometry(QtCore.QRect(120, 75, 161, 21))
```

```
        self.labelMessage.setText(_fromUtf8(""))
        self.labelMessage.setObjectName(_fromUtf8("labelMessage"))
        self.lineUserName = QtGui.QLineEdit(Dialog)
        self.lineUserName.setGeometry(QtCore.QRect(130, 30, 181, 20))
        self.lineUserName.setObjectName(_fromUtf8("lineUserName"))
        self.retranslateUi(Dialog)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

    def retranslateUi(self, Dialog):
         Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None,
QtGui.QApplication.UnicodeUTF8))
        self.ClickMeButton.setText(QtGui.QApplication.translate("Dialog", "Click Me",
None, QtGui.QApplication.UnicodeUTF8))
        self.labelEnterName.setText(QtGui.QApplication.translate("Dialog", "Enter your
name", None, QtGui.QApplication.UnicodeUTF8))
```


As stated earlier, the top-level object used in the application is `Dialog`, hence the `Ui_Dialog` class is created that stores the interface elements of our widget. The class has two methods, `setupUi()` and `retranslateUi()`. The `setupUi()` method is for setting up the widgets and their properties, and the `retranslateUi()` method is for translating the interface.

The next task is to connect slots and write code for the slots to perform processing. For this, you need to write another Python script and import the previous Python code to invoke the user interface design. Let's create the source file, name it `callwelcome.pyw`, and import the Python code `welcomemsg` in it. The code in `callwelcome.pyw` is shown here:

```
callwellcome.pyw
import sys
from welcomemsg import *
class MyForm(QtGui.QDialog):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.ui = Ui_Dialog()
        self.ui.setupUi(self)
        QtCore.QObject.connect(self.ui.ClickMeButton,  QtCore.SIGNAL('clicked()'),self.
dispmessage)

    def dispmessage(self):
        self.ui.labelMessage.setText("Hello "+ self.ui.lineUserName.text())

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    myapp = MyForm()
    myapp.show()
    sys.exit(app.exec_())
```

Figure 7.16
Welcome message displayed on selecting the Click Me button.

As stated earlier, every PyQt GUI application must have a `QApplication` object to provide access to information such as the application's directory, screen size, and so on. You create an instance of `MyForm` and call its `show()` method, which adds a new event to the `QApplication` object's event queue. The `app.exec_()` method is called to start the `QApplication` object's event loop. Once the event loop begins, the top-level widget used in the class `MyForm` is displayed, along with its child widgets. On occurrence of an event, the event loop processes it and eventually passes it to the associated method. To the `__init__()` method of `MyForm`, you pass a default parent of `None` to cause the `QDialog` class to be treated as a top-level window.

Recall that the user interface design is instantiated by calling the `setupUI()` method of the class that was created in the Python code (`Ui_Dialog`). You need to create an instance of the class `Ui_Dialog`, the class that was created in the Python code, and invoke its `setupUi()` method. On calling the `setupUi()` method, the Dialog widget will be created as the parent of all the widgets and displayed on the screen.

To respond to the events, the `clicked()` signal (event) of the Click Me pushbutton with the `ClickMeButton` object name is connected to the `dispmessage()` slot (method). Hence, when the user selects the Click Me pushbutton, the code in `dispmessage()` will be executed. The code in `dispmessage()` retrieves the name entered by the user in the Line Edit widget, `lineUserName`, and displays it through the Label `labelMessage` after prefixing it with a string, `Hello`. In Figure 7.16, you can see that if the user enters the user name **Caroline** in Line Edit and selects the Click Me pushbutton, the welcome displayed via Label will be `Hello Caroline`.

Converting Data Types

The default data type in a Line Edit widget is string. What if you want to use the widget for numerical data? Let's think of an application where you want to add two integer values and print their sum through a Label widget. First you need to convert string data entered in the Line Edit widget to integer data type and then convert the sum of the numbers, which will be of integer data type, back to string type before being displaying through a Label widget.

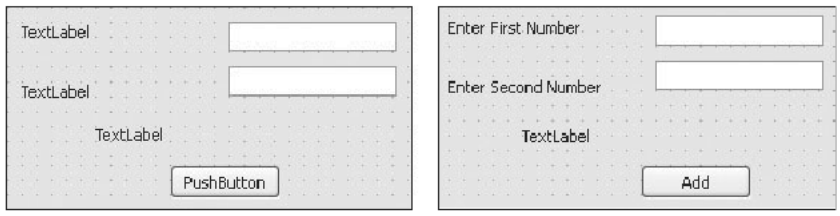


Figure 7.17
(a) Four widgets dropped on the form. (b) Widgets on the form with the `text` property set.

Let's create an application based on the Dialog without Buttons template and add three `QLabels`, two `QlineEdits`, and a `QPushButton` to the form by dragging and dropping three Labels, two Line Edits, and a Push Button on the form as shown in Figure 7.17(a). Set the `text` property of the two Label widgets to `Enter First Number` and `Enter Second Number` (Figure 7.17(b)). Set the `objectName` property of the three Labels to `labelFirstNumber`, `labelSecondNumber`, and `labelAddition`. Also, set the `objectName` property of the two Line Edit widgets to `lineFirstNumber` and `lineSecondNumber`. Set the `objectName` property of the Push Button to `AddButton` and also change its text property to `Add`. You don't need to change the third label's text property because the Python script will set the value and then display it when the two numerical values are added. Also, remember to drag the Label widget in the Designer to ensure it is long enough to display the text that will be assigned to it through the Python script. You can also increase the width of the Label widget by selecting Geometry > Width Property from the Property Editor.

Save the UI file as addtwonum.ui. The .ui file, which is in XML format when converted into Python code will appear as shown here:

```
addtwonum.py
# Form implementation generated from reading ui file 'addtwonum.ui'
from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s

class Ui_Dialog(object):
    def setupUi(self, Dialog):
        Dialog.setObjectName(_fromUtf8("Dialog"))
        Dialog.resize(435, 255)
```

```
        self.lineFirstNumber = QtGui.QLineEdit(Dialog)
        self.lineFirstNumber.setGeometry(QtCore.QRect(190, 30, 113, 20))
        self.lineFirstNumber.setObjectName(_fromUtf8("lineFirstNumber"))
        self.lineSecondNumber = QtGui.QLineEdit(Dialog)
        self.lineSecondNumber.setGeometry(QtCore.QRect(190, 70, 113, 20))
        self.lineSecondNumber.setObjectName(_fromUtf8("lineSecondNumber"))
        self.labelSecondNumber = QtGui.QLabel(Dialog)
        self.labelSecondNumber.setGeometry(QtCore.QRect(50, 70, 111, 16))
        self.labelSecondNumber.setObjectName(_fromUtf8("labelSecondNumber"))
        self.AddButton = QtGui.QPushButton(Dialog)
        self.AddButton.setGeometry(QtCore.QRect(180, 130, 75, 23))
        self.AddButton.setObjectName(_fromUtf8("AddButton"))
        self.labelFirstNumber = QtGui.QLabel(Dialog)
        self.labelFirstNumber.setGeometry(QtCore.QRect(60, 30, 101, 16))
        self.labelFirstNumber.setObjectName(_fromUtf8("labelFirstNumber"))
        self.labelAddition = QtGui.QLabel(Dialog)
        self.labelAddition.setGeometry(QtCore.QRect(100, 100, 171, 21))
        self.labelAddition.setObjectName(_fromUtf8("labelAddition"))
        self.retranslateUi(Dialog)
        QtCore.QMetaObject.connectSlotsByName(Dialog)

    def retranslateUi(self, Dialog):
        Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None,
QtGui.QApplication.UnicodeUTF8))
        self.labelSecondNumber.setText(QtGui.QApplication.translate("Dialog", "Enter
Second Number", None, QtGui.QApplication.UnicodeUTF8))
        self.AddButton.setText(QtGui.QApplication.translate("Dialog", "Add", None,
QtGui.QApplication.UnicodeUTF8))
        self.labelFirstNumber.setText(QtGui.QApplication.translate("Dialog", "Enter
First Number", None, QtGui.QApplication.UnicodeUTF8))
        self.labelAddition.setText(QtGui.QApplication.translate("Dialog", "TextLabel",
None, QtGui.QApplication.UnicodeUTF8))
```

Let's create a Python script named `calltwonum.pyw` that imports the Python code `addtwonum.py` to invoke a user interface design and that fetches the values entered in the Line Edit widgets and displays their addition. The code in the Python script `calltwonum.pyw` is shown here:

```
calltwonum.pyw
import sys
from addtwonum import *

class MyForm(QtGui.QDialog):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.ui = Ui_Dialog()
```

```
        self.ui.setupUi(self)
        QtCore.QObject.connect(self.ui.AddButton,  QtCore.SIGNAL('clicked()'),  self.
dispsum)

    def dispsum(self):
        if len(self.ui.lineFirstNumber.text())!=0:
            a=int(self.ui.lineFirstNumber.text())
        else:
            a=0
        if len(self.ui.lineSecondNumber.text())!=0:
            b=int(self.ui.lineSecondNumber.text())
        else:
            b=0
        sum=a+b
        self.ui.labelAddition.setText("Addition: " +str(sum))
if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    myapp = MyForm()
    myapp.show()
    sys.exit(app.exec_())
```


Before we look at the code, let's consider the three functions used in it:

`len()`: Returns the number of characters in the string.

`str()`: Converts the passed argument into string data type.

`int()`: Converts the passed argument into integer data type.

The `clicked()` event of `AddButton` is connected to the `dispsum()` method to display the sum of the numbers entered in the two Line Edits. In the `dispsum()` method, you first validate `lineFirstNumber` and `lineSecondNumber` to ensure that if either Line Edit is left blank by the user, the value of that Line Edit is zero. The value entered in the two Line Edits is retrieved, converted into integers through `int()`, and assigned to the two variables `a` and `b`. The sum of the values in the variables `a` and `b` is computed and stored in the variable `sum`. The result in the variable `sum` is converted into string format through `str()`and displayed via `labelAddition`. You can see in Figure 7.18 that when the user selects the Add button after entering two numbers in the Line Edits, the addition is displayed through the Label widget.

Can you have a shortcut key for Line Edits? Consider a form with several Line Edit widgets that you want to access with a shortcut key. It is possible through buddies.

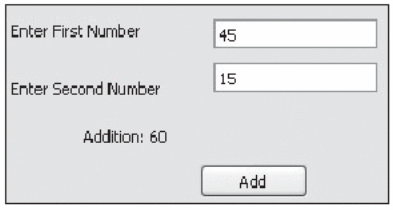


Figure 7.18
The sum of the numbers entered in Line Edit is displayed through a Label widget.

Defining Buddies

To establish a connection through between widgets or relate corresponding widgets, you create *buddies*. The benefit of using buddies is to have quick keyboard focus via shortcut keys on the widgets that do not accept focus. For example, to get focus on a Line Edit widget, you can set it as a buddy of a Label widget and assign a shortcut key to the Label widget. When the user presses the shortcut key for the Label widget, keyboard focus will be set on its Line Edit buddy widget.

Let's create a new application based on the Dialog without Buttons template. Add four `QLabels`, three `QlineEdits`, and a `QPushButton` to the dialog by dragging and dropping from the respective groups in the Widget Box. Set the `text` property of the three Label widgets to `&Number of items`, `&Price per item`, and `&Discount Percentage`. Recall that preceding any character in the text with an ampersand (`&`) will make it a shortcut key for the selected widget. Assigning the text `&Number of items` to the first Label (see Figure 7.19) will declare its first character, `N`, as its shortcut key. That also means that the Label will be accessed by

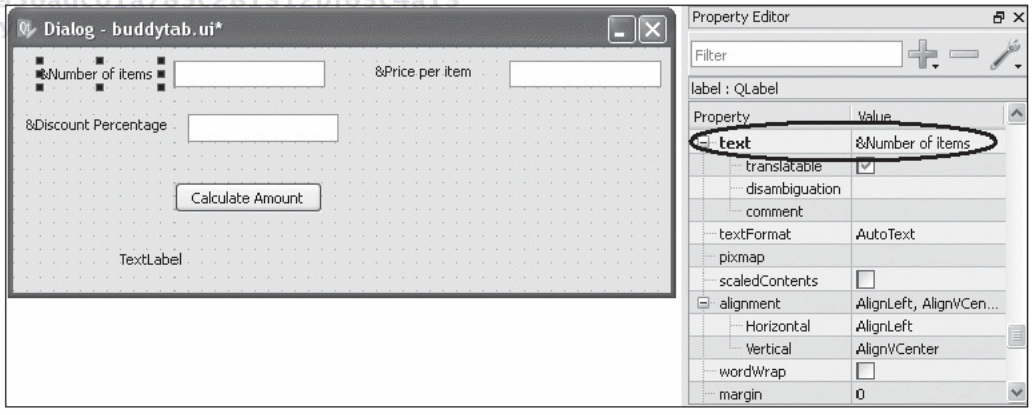


Figure 7.19
Setting a shortcut key for the Label.

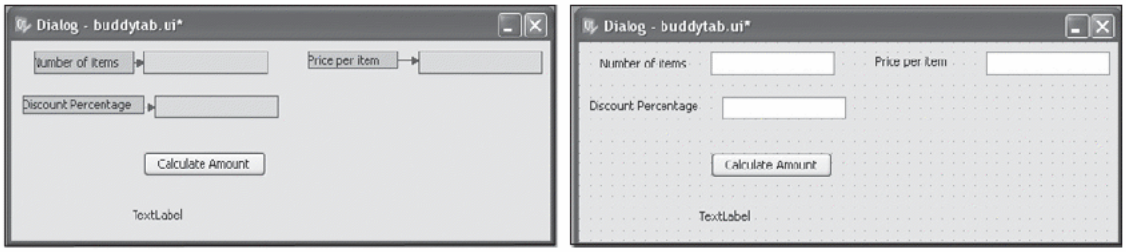


Figure 7.20
(a) Widgets on the form with the buddies set. (b) The dialog in Widget Editing mode.

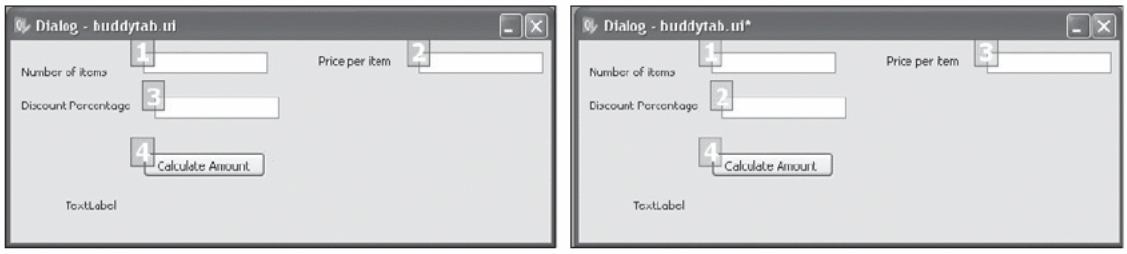
Alt+N. Similarly, the shortcut keys for the next two Labels will be Alt+P and Alt+D. Set the text for the button to **Calculate Amount**. Set the `objectName` of the three Line Edit widgets to `quantity`, `rate`, and `discount`. Recall that it is through the object names that the widgets are distinguished and accessed in coding. Also set the `objectName` of the fourth Label to `result` and leave its default `text` property `TextLabel` as such because you will be setting its actual text in the program for displaying the result of computation. Also, increase the width of the fourth label either by dragging its nodes in the Designer or by selecing Geometry > Width Property in the Property Editor so that it can display all the information assigned to it by the script.

To begin setting buddies, select Edit > Edit Buddies or the Edit Buddies icon from the toolbar to switch to Buddy Editing mode. To go back to Widget Editing mode from Buddy Editing mode, you can choose any of the three options: press F3, select the Edit > Edit Widgets from the menu, or click the Edit Widgets icon on the toolbar. In Buddy Editing mode, select a Label widget and drag it to the Line Edit widget that you want to set as its buddy and release the mouse button. The Label and Line Edit widgets will become buddies. On defining a buddy for the Label widget, the & (ampersand) in its text becomes invisible. After setting the three Line Edit widgets as buddies of the Label widgets, the dialog will appear as shown in Figure 7.20(a). To switch from Buddy Editing to Widget Editing mode, either press F3 or select the Edit Widgets icon from the toolbar. The dialog in Widget Editing mode will appear as shown in Figure 7.20(b).

Before running the application, let's see how to set the tab order of the widgets.

Setting Tab Order

Tab order means the order in which the widgets will get focus when the Tab and Shift+Tab keys are pressed. The default tab order is based on the order in which widgets are placed on the form. To change this order, you need to switch to Tab Order



(a) (b)

Figure 7.21
(a) Initial tab order of the widgets on the form. (b) Modified tab order of the widgets on the form.

Editing mode by either selecting the Edit, Edit Tab Order option or choosing the Edit Tab Order icon from the toolbar. In Tab Order Editing mode, each input widget in the form is shown with a number indicating its position in the tab order (see Figure 7.21(a)). If the user gives the first input widget the input focus and then presses the Tab key, the focus will move to the second input widget, and so on. You can change the tab order by clicking on each number in the correct order. When you select a number, it will change to red, indicating the currently edited position in the tab order chain. Clicking on the next number will make it the second in the tab order, and so on. In case of a mistake, you can restart numbering by choosing Restart from the form's context menu. To edit the tab order in the middle of the form, select a number with the Ctrl key pressed from where you want to change the tab order or choose Start from Here from the context menu. Let's set the tab order of the widgets on our dialog as shown in Figure 7.21(b).

Note

There is one more way to specify the tab order. Right-click anywhere on the form and select Tab Order List from the context menu that appears.

Save the application with the name `buddytab.ui`. Upon conversion to Python code, the XML file `buddytab.ui` will appear as shown here:

buddytab.py

```
# Form implementation generated from reading ui file 'buddytab.ui'

from PyQt4 import QtCore, QtGui

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    _fromUtf8 = lambda s: s
```

```
class Ui_Dialog(object):
  def setupUi(self, Dialog):
    Dialog.setObjectName(_fromUtf8("Dialog"))
    Dialog.resize(490, 182)
    self.label = QtGui.QLabel(Dialog)
    self.label.setGeometry(QtCore.QRect(10, 20, 91, 16))
    self.label.setObjectName(_fromUtf8("label"))
    self.quantity = QtGui.QLineEdit(Dialog)
    self.quantity.setGeometry(QtCore.QRect(120, 10, 113, 20))
    self.quantity.setObjectName(_fromUtf8("quantity"))
    self.label_2 = QtGui.QLabel(Dialog)
    self.label_2.setGeometry(QtCore.QRect(280, 10, 71, 16))
    self.label_2.setObjectName(_fromUtf8("label_2"))
    self.rate = QtGui.QLineEdit(Dialog)
    self.rate.setGeometry(QtCore.QRect(370, 10, 113, 20))
    self.rate.setObjectName(_fromUtf8("rate"))
    self.label_3 = QtGui.QLabel(Dialog)
    self.label_3.setGeometry(QtCore.QRect(10, 50, 101, 16))
    self.label_3.setObjectName(_fromUtf8("label_3"))
    self.discount = QtGui.QLineEdit(Dialog)
    self.discount.setGeometry(QtCore.QRect(130, 50, 113, 20))
    self.discount.setObjectName(_fromUtf8("discount"))
    self.pushButton = QtGui.QPushButton(Dialog)
    self.pushButton.setGeometry(QtCore.QRect(120, 100, 111, 23))
    self.pushButton.setObjectName(_fromUtf8("pushButton"))
    self.result = QtGui.QLabel(Dialog)
    self.result.setGeometry(QtCore.QRect(50, 140, 351, 16))
    self.result.setText(_fromUtf8(""))
    self.result.setObjectName(_fromUtf8("result"))
    self.label.setBuddy(self.quantity)
    self.label_2.setBuddy(self.rate)
    self.label_3.setBuddy(self.discount)
    self.retranslateUi(Dialog)
    QtCore.QMetaObject.connectSlotsByName(Dialog)
    Dialog.setTabOrder(self.quantity, self.discount)
    Dialog.setTabOrder(self.discount, self.rate)
    Dialog.setTabOrder(self.rate, self.pushButton)

  def retranslateUi(self, Dialog):
      Dialog.setWindowTitle(QtGui.QApplication.translate("Dialog", "Dialog", None,
QtGui.QApplication.UnicodeUTF8))
     self.label.setText(QtGui.QApplication.translate("Dialog", "&Number of items",
None, QtGui.QApplication.UnicodeUTF8))
    self.label_2.setText(QtGui.QApplication.translate("Dialog", "&Price per item",
None, QtGui.QApplication.UnicodeUTF8))
```

```
        self.label_3.setText(QtGui.QApplication.translate("Dialog", "&Discount Per-
centage", None, QtGui.QApplication.UnicodeUTF8))
         self.pushButton.setText(QtGui.QApplication.translate("Dialog", "Calculate
Amount", None, QtGui.QApplication.UnicodeUTF8))
```

Let's create a Python script to import the Python code to invoke the user interface design and to compute the amount when `number of items`, `price per item`, and `discount percentage` are supplied by the user. Name the Python script `callbuddytab.pyw`; its code is shown below:

```
callbuddytab.pyw
from __future__ import division
import sys
from buddytab import *

class MyForm(QtGui.QMainWindow):
   def __init__(self, parent=None):
      QtGui.QWidget.__init__(self, parent)
      self.ui = Ui_Dialog()
      self.ui.setupUi(self)
      QtCore.QObject.connect(self.ui.pushButton,  QtCore.SIGNAL('clicked()'),  self.
calculate)

   def calculate(self):
      if len(self.ui.quantity.text())!=0:
         q=int(self.ui.quantity.text())
      else:
         q=0
      if len(self.ui.rate.text())!=0:
         r=int(self.ui.rate.text())
      else:
         r=0
      if len(self.ui.discount.text())!=0:
         d=int(self.ui.discount.text())
      else:
         d=0
      totamt=q*r
      disc=totamt*d/100
      netamt=totamt-disc
      self.ui.result.setText("Total Amount: " +str(totamt)+", Discount: "+str(disc)+",
Net Amount: "+str(netamt))

if __name__ == "__main__":
   app = QtGui.QApplication(sys.argv)
   myapp = MyForm()
   myapp.show()
   sys.exit(app.exec_())
```

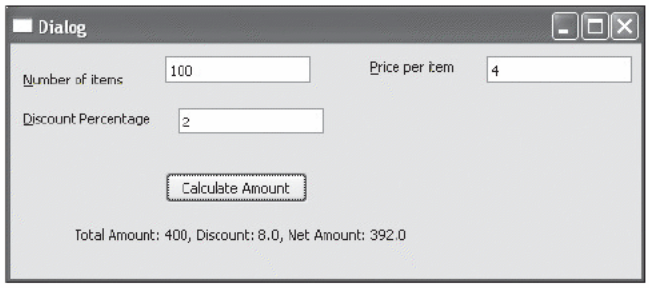


Figure 7.22
The characters acting as shortcut key appear underlined.

In this code, you can see that the Push Button's `clicked()` signal is connected to the `calculate()` function. After supplying the values for `number of items`, `price per item`, and `discount percentage` in the Line Edit widgets, when the user selects the Calculate Amount Push Button, the `calculate()` function will be invoked. In the `calculate()` function, you validate the Line Edit widgets to check if any Line Edit widget is left blank. The value of Line Edit that is left blank is assumed to be 0. Thereafter, you compute the net amount, which is `total amount` minus `discount`, where `total amount` is the product of `number of items` and `price per item`. The computed `net amount` is then converted to string data type to be displayed via a Label widget.

On running the application, you will find the underscored characters `N`, `P`, and `D` in the Label's texts, Number of items, Price per item, and Discount Percentage, as shown in Figure 7.22. The underscored characters mean that you can use Alt+N, Alt+P, and Alt+D shortcut keys for setting focus to the respective Line Edit widgets for entering values for `number of items`, `price per item`, and `discount percentage`. If you don't see the underscored characters in the Labels, just press Alt, and underscores will appear.

Summary

In this chapter you had a brief introduction to the Qt toolkit and PyQt. You learned the procedure of installing PyQt. You learned about different Qt Designer components such as the toolbar, the Object Inspector, the Property Editor, and the Widget Box. You also learned to create a GUI application through coding. You learned about the fundamental Label, Line Edit, and Push Button widgets and developed applications using them.

You also had a good introduction to signal/slot connections in Qt Designer and learned to connect signals to the predefined slots and to custom slots. In the next chapter you will learn about basic widgets such as Radio Buttons, Checkboxes, Spin Boxes, Scroll Bars, Sliders, and Lists. To better understand these basic widgets, you will develop individual application using each of them.