Appendix: Reference Guide for PyQt6

PyQt is a Python binding for the Qt Application framework maintained by Riverbank Computing Limited. A **binding** is an application programming interface (API) that provides the code to allow a programming language to use other libraries not native to that language. Qt is a set of C++ libraries and development tools, providing access to networking, threads, SQL databases, OpenGL and other graphics tools, XML, GUI development, and a variety of other features. This chapter focuses only on PyQt6, but many of the concepts and methods are still available in PyQt5.

Appendix contains a reference for some of the tools, modules, and classes learned throughout this book, including

- A review of PyQt modules and classes
- An overview of Qt Style Sheets
- A discussion about Qt Namespace

More information about Riverbank Computing Limited and PyQt6 can be found at `https://riverbankcomputing.com/software/pyqt/intro`.

Selected PyQt6 Modules

PyQt provides a range of modules that give you access to a wide array of tools, including basic GUI design, 2D and 3D rendering, multimedia content, networking, global positioning, and more. For basic GUI development, you will primarily use the `QtWidgets`, `QtGui`, and `QtCore` modules. Table A-1 lists the modules covered throughout the book as well as a few extra you should check out.

For a full list of PyQt6's top-level modules, check out the following link:

Table A-1 Table of select PyQt modules

Module Name	**Description**
`QtWidgets`	Provides the widgets and other classes for creating desktop-style UIs
`QtCore`	Contains a variety of extra classes, including the essential non-GUI classes, such as ones for Qt's signal and slot system

Module Name	Description
QtGui	Contains classes for 2D graphics and imaging, event handling, and window system integration
QtPrintSupport	Provides cross-platform support for configuring and connecting to printers
QtNetwork	Provides classes for writing communications protocols using UDP or TCP
QtQuick	Contains the classes for creating QML applications with Python
QtMultimedia	Contains the classes for multimedia content, including cameras, images, and audio
QtMultimediaWidgets	Provides additional classes that increase the functionality of multimedia-related widgets
QtWebEngineCore	Contains the core classes used by other Web Engine modules
QtWebEngineWidgets	Classes that can be used to create a Chromium-based web browser
QtSql	Provides classes for working with SQL databases
sip	Tools used for creating Python bindings for C ++ libraries (which is the language Qt is written in)
uic	Contains classes used for handling the .ui files created by Qt Designer

www.riverbankcomputing.com/static/Docs/PyQt6/module_index.html

Selected PyQt Classes

There are hundreds of PyQt classes. The following section lists the classes and widgets that can be found throughout this book. Each subsection either lists tables with commonly used methods and signals or a link to where you can find more information about the class.

For a list of all the PyQt classes, check out

www.riverbankcomputing.com/static/Docs/PyQt6/sip-classes.html

Although it is written for C++, the Qt classes documentation is generally more detailed. If you want more information about Qt classes, you can also check out https://doc.qt.io/qt-6/classes.html

Just keep in mind that some of the classes that exist in Qt are not available in PyQt. In many cases, this is because Python already includes the functionality that the removed class would provide. One common example is `QList`, which exists in Qt but is not included in Python since it includes the `list` data structure.

Classes for Building a GUI Window

With PyQt, you can create a new class that inherits from any of the widget classes. However, for a general GUI application, you will need to create only one instance of `QApplication` and create a class that inherits from either `QWidget`, `QMainWindow`, or `QDialog` to create the application's main window.

QApplication

`QApplication` is responsible for handling the initialization and finalization of widgets in graphical user interfaces. If you are making `QWidget`-based applications, then you will need to create an instance of `QApplication` before creating any other objects related to the GUI.

Some of the `QApplication` class's responsibilities include initializing an application to conform to a user's desktop settings, event handling, defining the GUI's style, working with the clipboard, and keeping track of all the application's windows.

If you are creating an application that does not need a GUI and can be run through the command line, then you should consider using **`QCoreApplication`** instead.

QWidget

The `QWidget` class is the base class for all of PyQt's graphical user interface objects. A widget created from the `QWidget` class is able to receive input from mouse, keyboard, and other events and able to paint itself on the screen. Widgets that are not embedded in a parent widget are considered to be a window complete with a title bar and a frame. The `QWidget` class is a subclass of `QObject` and **`QPaintDevice`** (the class that defines a two-dimensional space for drawing on with `QPainter`). Some helpful `QWidget` methods can be found in Table A-2.

Table A-2 Selected methods from QWidget

Method	Description
`addAction(action)`	Adds an `action` to the widget

Method	Description
`close()`	Closes the widget
`height()`	Retrieves the widget’s height
`width()`	Retrieves the widget’s width
`move(x, y)`	Sets the location of the widget to (`x`, `y`)
`rect()`	Retrieves the geometry of the widget minus the frame
`setDisabled(bool)`	If `True`, the widget is disabled
`setEnabled(bool)`	If `True`, the widget is enabled
`setFont(font)`	Sets the `font` of the widget’s text (if the widget can display text)
`setLayout(layout)`	Sets the layout manager for the widget
`setGeometry(x, y, width, height)`	Sets the widget’s location, (`x`, `y`), and its size, `width` and `height`
`setStyleSheet(styleSheet)`	Sets the `styleSheet` for the widget
`setToolTip(text)`	Sets the widget’s tool tip
`repaint()`	Repaints the widget immediately by calling `paintEvent()`
`showFullScreen()`	Displays the widget in full-screen mode
`update()`	Updates the widget by scheduling a paint event in the main event loop

Event Handling

Events are typically caused by users or the underlying system. These can include moving a mouse, pressing a key, resizing the window, or a timer delivering events. The widgets in an application need to respond appropriately to the event. The events are generally already handled in the background of simpler applications, but you sometimes may find yourself needing to reimplement event handlers to supply further behavior or content for the widgets. Table A-3 lists a few commonly used event handlers.

Table A-3 Some event handlers used for supplying behavior to QWidget objects

Event Handler	Description
`paintEvent()`	Called whenever a widget needs to be repainted
`resizeEvent()`	Called when a widget has been resized

Event Handler	Description
`mousePressEvent()`	Called when a mouse button is pressed while the mouse cursor is inside of the widget. Which mouse button is clicked can be specified in the event
`mouseReleaseEvent()`	Called when a mouse button is released. A widget that receives this event is dependent on receiving the mouse press event
`mouseDoubleClickEvent()`	Called when a widget is double-clicked on
`mouseMoveEvent()`	Called when the mouse moves while the button is held down. If `setMouseTracking()` is `True`, events are sent even when no buttons are pressed
`enterEvent()`	Called when the mouse enters a widget's space
`leaveEvent()`	Called when the mouse leaves a widget's space
`keyPressEvent()`	Called when a key is pressed
`keyReleaseEvent()`	Called when a key is released
`focusInEvent()`	Called when a widget gets the keyboard focus
`focusOutEvent()`	Called when a widget loses the keyboard focus
`closeEvent()`	Called when either a widget or the window is closed

QMainWindow

The `QMainWindow` class provides the framework for building an application, complete with functions for adding a menu bar, toolbars, a status bar, and dock widgets. Menu and toolbar items are created using `QAction`. `QMainWindow` already has its own layout, to which you must add a central widget as the center area of the application's window. Some of the `QMainWindow` class's methods can be seen in Table A-4.

Table A-4 Select methods from QMainWindow

Method	Description
`addDockWidget(area, dockwidget)`	Creates a `dock widget` in the main window in the specified `area`
`addToolBar(area, toolbar)`	Creates a `toolbar` for the main window. An `area` can also be specified
`menuBar()`	Returns the main window's menu bar
`setStatusBar(statusbar)`	Creates the `status bar` for the main window
`setCentralWidget(widget)`	Sets the window's central `widget`
`setWindowIcon(icon)`	Sets the window's `icon`

Method	Description
`setWindowTitle(text)`	Sets the window's title. This is a method inherited from `QWidget`

QDialog

Dialog boxes provide a top-level window that are generally used to quickly obtain feedback from a user. QDialog instances can be modal or modeless. Modal dialogs are often used when selecting an option in the dialog that will return a value. That value could then be used to save a file, close a document, or cancel an action.

`QDialog` is the base class for other dialog box classes, including `QColorDialog`, `QFileDialog`, `QFontDialog`, `QInputDialog`, `QMessageBox`, `QProgressDialog`, and `QErrorMessage`. A few methods for setting the mode of the dialog and handling the results of the dialog are in Table A-5.

Table A-5 Select methods for QDialog

Method	Description
`accept()`	Hides the modal dialog and returns `True`, accepting the actions specified by the dialog
`reject()`	Hides the modal dialog and returns `False`, rejecting the actions specified by the dialog
`open()`	The dialog is shown as a modal dialog and blocks the user from any further action until the dialog is closed
`show()`	The dialog is a modeless dialog, returning control to the user immediately

Table A-6 lists some common default buttons that are part of the `QMessageBox.StandardButton` or `QDialogButtonBox.StandardButton` enums. These flags are very useful when creating custom dialog boxes. Each one of the buttons returns a specific **ButtonRole**, describing the behavior of the button. For example, **`AcceptRole`** causes the dialog and its contents to be accepted. This is equivalent to OK. A **`RejectRole`** rejects the dialog, which is what Cancel does. There are other kinds of roles too. Refer to the table for more information.

Table A-6 Select standard buttons for QDialogButtonBox and QMessageBox

Method	Description
Ok	Defines an OK button with an AcceptRole
Open	Defines an Open button with an AcceptRole
Save	Defines a Save button with an AcceptRole
Cancel	Defines a Cancel button with a RejectRole
Close	Defines a Close button with a RejectRole
Yes	Defines a Yes button with a YesRole
No	Defines a No button with a NoRole
Reset	Defines a Reset button with a ResetRole

QPainter

The QPainter class is responsible for handling drawing in PyQt, being able to draw simple lines and complex shapes onto widgets and other paint devices. QPainter is most commonly used in the paintEvent() event handler, as well as for handling pixmaps and images. Table A-7 displays some of the QPainter class's methods for drawing.

Table A-7 Methods selected from QPainter

Method	Description
begin(device)	Begins painting on the paint device
end()	Ends painting. Resources used while painting are released
save()	Saves the current painter state. save() must be followed by restore(), which returns the current painter state
drawArc(QRectF, startAngle, spanAngle)	Draws an arc defined by the QRectF rectangle, startAngle, and spanAngle
drawChord(QRectF, startAngle, spanAngle)	Draws a chord defined by the QRectF rectangle, startAngle, and spanAngle
drawEllipse(QPointF, x_rad, y_rad)	Draws an ellipse at QPointF center, with radius x_rad and y_rad
drawLine(x1, y1, x2, y2)	Draws a line from point (x1, y1) to (x2, y2)
drawPath(path)	Draws a path specified by QPainterPath path
drawPie(QRectF, startAngle, spanAngle)	Draws a pie defined by the QRectF rectangle, startAngle, and spanAngle

Method	**Description**
`drawPixmap(x, y, pixmap)`	Draws a `pixmap` at (`x`, `y`)
`drawPoint(x, y)`	Draws a point at (`x`, `y`)
`drawRect(x, y, width, height)`	Draws a rectangle at (`x`, `y`) with `width` and `height`
`drawRoundedRect(QRectF, x_rad, y_rad)`	Draws a rectangle with rounded corners specified by `QRectF`, with radius `x_rad` and `y_rad`
`drawText(QPointF, text)`	Draws `text` at `QPointF` point
`fillRect(QRectF, brush)`	Fills in a `QRectF` rectangle with the `brush` color
`rotate(angle)`	Rotates the coordinate system clockwise by `angle` (in degrees)
`setBrush(brush)`	Sets the painter's `brush`
`setPen(pen)`	Sets the painter's `pen`
`setFont()`	Sets the painter's `font`

Layout Managers

Using PyQt's layout managers makes the process of arranging widgets much easier compared to manually specifying each widget's size, position, or `resizeEvent()` event handler. Using layout managers is generally a good start for positioning widgets, although you may still need to adjust a widget's size policy or add stretching or spacing to a layout.

The following classes inherit from the **`QLayout`** class, which is the base class for layout managers:

1. `QBoxLayout` – Arranges child widgets into a row (horizontally) or into a column (vertically)
    a. `QHBoxLayout` – Arranges widgets horizontally
    b. `QVBoxLayout` – Arranges widgets vertically

2. `QGridLayout` – Orders widgets in a grid of rows and columns
3. `QFormLayout` – Lays out widgets into a form-like structure with labels and their associated input widgets
4. `QStackedLayout` – Arranges widgets into a stack where only one widget is visible at a time. The convenience `QStackedWidget` class

is built on top of the `QStackedLayout`.

Table A-8 lists commonly used methods from the layout classes.

Table A-8 Selected methods for the different layout managers

Method	Class	Description
`addWidget(widget, stretch, alignment)`	`QBoxLayout`	Adds `widget` to the end of the layout with `stretch` factor and `alignment`
`addWidget(widget, row, column, rowSpan, columnSpan alignment)`	`QGridLayout`	Adds `widget` at `row`, `column` with (optional) `rowSpan` and `columnSpan` and `alignment`
`addRow(label, field)`	`QFormLayout`	Adds a new row with a given `label` and `field` (input widget)
`addWidget(widget)`	`QStackedLayout`	Adds a new `widget` to the end of the layout. This method returns the widget's index in the stack
`addLayout(layout, stretch)`	`QBoxLayout`	Adds a `layout` to the end of the box. Creates a nested layout
`addLayout(layout, row, column, alignment)`	`QGridLayout`	Adds a `layout` at position (`row`, `column`). Creates a nested layout
`addSpacing(int)`	`QGridLayout`, `QBoxLayout`	Adds a nonstretchable area (a `QSpacerItem`) of `int` value to the layout
`addStretch(int)`	`QBoxLayout`	Adds a stretchable area (a `QSpacerItem`) of `int` value to the layout
`setSpacing(int)`	`QLayout`	Sets the space between widgets in the layout. Inherited from `QLayout`
`setContentMargins(left, top, right, bottom)`	`QLayout`	Sets the `left`, `top`, `right`, and `bottom` margins around the layout

Button Widgets

Buttons are one of the main tools used in a GUI for interaction, giving an application feedback about a user's decisions. Buttons in PyQt can display text or icons and are checkable. The following classes inherit from the base class for button widgets, **`QAbstractButton`**:

1. `QPushButton` – A command button used to tell the computer to

perform some action

2. QCheckBox – Provides an option button that is checkable and generally used for enabling/disabling features in an application
3. QRadioButton – Similar to check boxes, but are mutually exclusive
4. QToolButton – Typically used in a toolbar, tool buttons provide quick-access buttons for selecting commands or options

For managing and organizing multiple buttons, the QButtonGroup class can act as a container for creating exclusive buttons (the default setting). Table A-9 lists some of the more commonly used methods for button widgets.

Table A-9 Selected methods for the different button widgets

Method	Description
setIcon(icon)	Sets the widget's icon
setText(text)	Sets the widget's text
setAutoExclusive(bool)	Enables autoexclusivity for buttons in a group
setCheckable(bool)	Sets whether the button is a toggle button or not
setChecked(bool)	Sets whether the button is checked or not
isChecked()	Indicates whether the button is checked or not (if setCheckable() is True)
text()	Gets the buttons text

Some signals for the button widget classes are listed in Table A-10.

Table A-10 Signals for the different button widgets

Signal	Class	Description
clicked(bool)	QAbstractButton	Signal emitted when the button is pressed and released
pressed()	QAbstractButton	Emitted when the left mouse button clicks on the button
released()	QAbstractButton	Signal emitted when the left mouse button is released
toggled(bool)	QAbstractButton	Emitted when a checkable button changes its state

Signal	Class	Description
`stateChanged(bool)`	`QCheckBox`	Emitted when the check box's state changes
`triggered(action)`	`QToolButton`	Signal emitted when the `action` is triggered

Input Widgets

There are quite a few widgets that are provided by PyQt for getting input from the user. These widgets provide different means for gathering information, such as text entry or selecting values with sliders, combo boxes, and spin boxes.

Combo Boxes

The `QComboBox` class presents a user with a list of selectable options in a compact, drop-down menu. Some of the class's methods are found in Table A-11. When the combo box is not being interacted with, all items except for the current item selected are hidden from view. The **`QFontComboBox`** widget is another type of combo box that inherits `QComboBox` and is used for selecting a font family.

Table A-11 Select methods from the QComboBox class

Method	Description
`addItem(text)`	Appends an item to the list with `text`
`addItems(list(text))`	Appends a `list` of items to the combo box
`currentIndex()`	Gets the index of the currently selected item
`currentText()`	Gets the text of the currently selected item
`insertItem(index, text)`	Inserts the `text` into the combo box at the given `index`
`setItemText(index, text)`	Sets the `text` for the item at the given `index`
`removeItem(index)`	Removes the item at the given `index`
`clear()`	Clears all items from the combo box
`setEditable(bool)`	If `True`, the contents of the combo box are editable

Table A-12 displays select signals for the combo box classes.

Table A-12 Commonly used signals from the QComboBox and QFontComboBox classes

Signal	Description

Signal	Description
`currentIndexChanged(index)`	Emitted if the current item in the combo box has changed
`currentTextChanged(text)`	Signal emitted if the current item in the combo box has changed. Returns `text`
`activated(index)`	Emitted only if the user interacts with an item
`highlighted(index)`	Emitted when an item in the combo box is highlighted
`textActivated(text)`	Signal emitted when the user chooses an item
`currentFontChanged(font)`	Emitted when the current `font` changes

QLineEdit

The `QLineEdit` widget provides a single line for entering and editing plain text. Although not listed in the following tables, `QLineEdit` includes `clear()`, `selectAll()`, `cut()`, `copy()`, `paste()`, `undo(),` and `redo()` slots already built-in. Table A-13 displays a few of the `QLineEdit` class's methods.

Table A-13 Methods from the QLineEdit class

Method	Description
`text()`	Retrieves the current text in the line edit
`setAlignment(alignment)`	Sets the `alignment` of the text displayed in the widget
`setPlaceholderText(text)`	Displays placeholder `text` while line edit is empty
`setEchoMode(mode)`	The parameter `mode` describes how the contents of a line should be displayed. Set `mode` to `QLineEdit.Password` to mask characters
`setMaxLength(int)`	Sets the maximum length of characters
`setTextMargins(left, top, right, bottom)`	Sets the text margins for the text displayed in the line edit
`setDragEnabled(bool)`	If `True`, dragging selected text in the line edit is permitted

A few common signals for `QLineEdit` can be seen in Table A-14.

Table A-14 Commonly used signals from the QLineEdit class

Signal	Description
`returnPressed()`	Emitted when the Enter key is pressed. If a validator is set, then a signal is only emitted if the text is accepted

Signal	Description
textChanged(text)	Signal is emitted when the text changes

Text Editing Widgets

The two text editing classes, QTextEdit and **QPlainTextEdit**, provide tools and functionality for displaying and editing larger bodies of text. QTextEdit also has the added benefit of being able to work with rich text, graphics, and tables. Both classes are similar to QLineEdit, because they already have editing features built-in. A few methods for text editors are found in Table A-15.

Table A-15 Select methods from QTextEdit and QPlainTextEdit

Method	Description
find(text, flags)	Finds the next occurrence of text in the text edit
print(printer)	Prints the text edit's document to the printer
setPlaceHolderText(text)	Sets placeholder text for text edit
setReadOnly(bool)	If True, the text edit is set to read-only
toPlainText()	Returns the text of the text edit as plain text
zoomIn(range)	Zooms in on the text
zoomOut(range)	Zooms out on the text

Also worth noting is the **QTextBrowser** class, which inherits QTextEdit. QTextBrowser only allows read-only mode but includes hypertext navigation functionality so that users can click on links and follow them.

Commonly used signals for the text editing widgets can be found in Table A-16.

Table A-16 Select signals from QTextEdit and QPlainTextEdit

Signal	Description
selectionChanged()	Signal emitted when the text selected in the text edit changes
textChanged()	Emitted whenever the contents of the text edit change

Spin Box Widgets

Spin boxes allow users to choose values within a given range by clicking up/down buttons to cycle through the widget's values. Users can also manually type in values into the provided line edit. The **`QAbstractSpinBox`** class is the base class for the following classes:

1. `QSpinBox` – Handles integers.
2. `QDoubleSpinBox` – Similar to `QSpinBox`, but is used for floating-point values.
3. `QDateTimeEdit` – A spin box widget for selecting dates and times. Use `setDisplayFormat()` to set the format used for displaying the dates and time.
4. `QDateEdit` – A spin box that displays only dates. Inherits `QDateTimeEdit`.
5. `QTimeEdit` – A spin box that displays only times. Inherits `QDateTimeEdit`.

Some of the methods for the `QSpinBox` and `QDoubleSpinBox` classes are listed in Table A-17. The `QDateTimeEdit` and other spin box widgets have similar methods.

Table A-17 Select signals from QSpinBox and QDoubleSpinBox. The value `val` refers to integers for QSpinBox and floating-point numbers for QDoubleSpinBox

Method	**Description**
`setValue(val)`	Sets the value `val` of the spin box
`setMinimum(val)`	Sets the minimum value `val` of the spin box
`setMaximum(val)`	Sets the maximum value `val` of the spin box
`setPrefix(str)`	Adds a prefix to the start of the displayed value
`setSuffix(str)`	Adds a suffix to the end of the displayed value
`setRange(min, max)`	Sets the minimum and maximum range values
`setSingleStep(val)`	The spin box's value is incremented/decremented by `val` when the arrow keys are pressed

Some `QSpinBox` and `QDoubleSpinBox` signals are found in Table A-18.

Table A-18 Signals from QSpinBox and QDoubleSpinBox

Signal	Description
`valueChanged(val)`	Signal emitted when the value changes. Provides the new value's `val`
`textChanged(text)`	Signal emitted when the value changes. Provides the new value's `text`

Slider Widgets

The following widgets are different in appearance but are actually quite similar in functionality. Widgets that inherit from the **`QAbstractSlider`** class are used for selecting integer values within a bounded range. Classes that inherit `QAbstractSlider` include the following:

1. `QDial` – Provides a rounded range controller for selecting or adjusting values. An example of `QDial` can be seen in Figure A-1.
2. `QScrollBar` – Provides horizontal or vertical scrollbars that the user can use to access other parts of a document that are wider than the widget used to display it.
3. `QSlider` – Creates the classic horizontal and vertical sliders widgets for controlling values within a specified range.

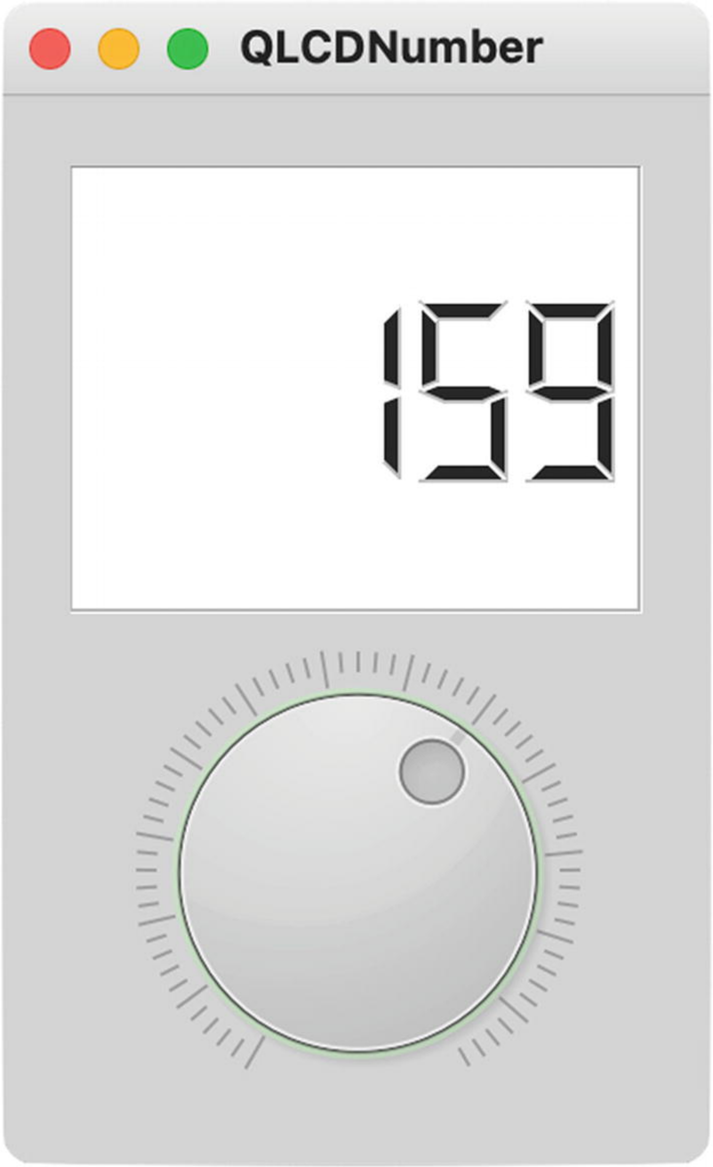

Figure A-1 Example of the QLCDNumber and QDial widgets. The XML and Python code for this example can be found in the Appendix folder of the GitHub repository

Table A-19 shows some of the methods of the `QAbstractSlider` base class.

Table A-19 Select methods from QAbstractSlider

Method	**Description**
`value()`	Holds the slider's current value
`setMinimum(int)`	Sets the minimum value of the slider
`setMaximum(int)`	Sets the maximum value of the slider

Method	Description
setOrientation(orientation)	Sets the orientation, Horizontal or Vertical (provided by the Qt.Orientation enum)
setSingleStep(int)	The slider’s value is incremented/decremented by int when the arrow keys are pressed
setTracking(bool)	If True, the slider’s position can be tracked
setSliderPosition(int)	Sets the current position of the slider
setValue(int)	Sets the current position of the slider to int. If tracking is enabled, then this has the same value as the value() getter

Signals of the QAbstractSlider class can be found in Table A-20.

Table A-20 Signals from QAbstractSlider

Signal	Description
valueChanged(val)	Signal emitted when the value changes. Provides the new value’s val
rangeChanged(min, max)	Signal emitted when the range has changed with new minimum and maximum values
sliderMoved(val)	Emitted when the slider is pressed down and the slider moves
sliderPressed()	Emitted when the slider is pressed down
sliderReleased()	Emitted when the slider is released

Display Widgets

The following widgets are all used for different purposes, but each has one major characteristic in common – they are all used for displaying information to the user.

QLabel

QLabel is a versatile widget. Although a label provides no user interaction functionality, QLabel is able to display plain or rich text, pixmaps, and even GIFs. Labels provide a number of methods for configuring their appearance. Table A-21 lists a few of those methods.

Table A-21 Select methods from QLabel

Method	Description
setPicture(picture)	Sets the label content to picture

Method	Description
`setPixmap(pixmap)`	Sets the label content to `pixmap`
`setMovie(movie)`	Sets the label content to `movie`
`setText(text)`	Sets the label content to `text`
`setAlignment(alignment)`	Sets the alignment of the label's content
`setIndent(int)`	Sets the number of pixels that the label's text is indented
`setMargin(int)`	Sets the label's margins

QProgressBar

Progress bars are used to give visual feedback to the user about the progress of a computer operation. Progress bars can be displayed vertically or horizontally. Table A-22 shows some of the `QProgressBar` class's methods.

Table A-22 Select methods for the QProgressBar class

Method	Description
`value()`	Holds the progress bar's current value
`setMinimum(int)`	Sets the progress bar's minimum value
`setMaximum(int)`	Sets the progress bar's maximum value
`setRange(min, max)`	Sets the minimum and maximum values
`setOrientation(orientation)`	Sets the `orientation`, `Horizontal` or `Vertical` (provided by the `Qt.Orientation` enum)
`setTextVisible(bool)`	If `True`, the current completed percentage is displayed

`QProgressBar` has one signal, `valueChanged(int)`, that is emitted when the value shown in the progress bar changes.

QGraphicsView

The `QGraphicsView` class provides a widget for displaying the contents of a `QGraphicsScene`. As the one part of Qt's Graphics View Framework, the `QGraphicsView` class's responsibility is to display the items of a graphics scene in a scrollable window. The `QGraphicsScene` object's duty is to manage the items in a scene. `QGraphicsItem` (or one of its subclasses) provides the items for a scene.

If you are interested in learning more about the Graphics View Framework, check out https://doc.qt.io/qt-6/graphicsview.html.

QLCDNumber

The **QLCDNumber** widget displays numbers in a seven-segment LCD display. An example of this is shown in Figure A-1. The display can visualize decimal, hexadecimal, octal, and binary numbers. The LCD display can only display certain characters. Note that if a character is passed that the widget cannot display, a space will be presented in place of the character.

Table A-23 lists a few of QLCDNumber class's methods.

Table A-23 Select methods from the QLCDNumber class

Method	Description
value()	Retrieves the LCD's displayed value
intValue()	Retrieves the displayed value rounded to the nearest integer value
display(val)	Displays the value val in the display. val can be floating-point, integer, or string types
setMode(mode)	Sets the mode of the LCD to display Bin, Oct, Dec, or Hex values
setSmallDecimalPoint(bool)	If True, the decimal is drawn between two digits

QLCDNumber has the overflow() signal, which is emitted when the widget is asked to display a number or string that is too long.

Item Views

The following model view classes provide the means to display items in lists, tables, or tree structures. They must be used alongside a model class as part of Qt's Model/View framework.

1. QListView – Provides a list and icon view for displaying items from a model
2. QTableView – Provides a table for displaying items from a model
3. QTreeView – Provides a hierarchical tree architecture for displaying items from a model

These classes all inherit from the **QAbstractItemView** class. Using signals and slots, item views created from QAbstractItemView are able to interact with models that use **QAbstractItemModel**. Each of the item views has their own methods for working with rows, columns, headers, and items. Views use indices to manage items. You can find some methods for QAbstractItemView in Table A-24.

Table A-24 Select methods for the QAbstractItemView base class

Method	Description
clearSelection()	All items selected are deselected
selectAll()	Selects all the items in the view
setCurrentIndex(index)	Sets the item at index as the current item
update(index)	Updates the area at the given index
setAlternatingRowColors(bool)	If True, the background is drawn with alternating colors
setAcceptDrops(bool)	If True, items can be dropped into the view
setDragEnabled(bool)	If True, items can be dragged around in the view
setIconSize(size)	Sets the size of icons
setItemDelegate(delegate)	Sets an item delegate for the view's Model/View framework
setModel(model)	Sets the model for the view

PyQt also offers convenience item-based classes for each of the different types of views – QListWidget, QTableWidget, and QTreeWidget. Items are added to the widgets by using QListWidgetItem, QTableWidgetItem, or QTreeWidgetItem.

Select signals for QAbstractItemView can be found in Table A-25.

Table A-25 Select methods for the QAbstractItemView base class

Signal	Description
activated(index)	Signal emitted when the item at index is activated by the user
clicked(index)	Emitted when the left mouse button is clicked on an item in the view (specified by index)
doubleClicked(index)	Emitted when a mouse button is double-clicked on an item in the view (specified by index)

Signal	Description
`entered(index)`	Signal emitted when the mouse cursor enters the item at `index`. Turn on mouse tracking to use
`pressed(index)`	Signal emitted when a mouse button is pressed on an item at `index`

Container Widgets

PyQt provides a few container widgets for maintaining control over groups of widgets. Containers can be used to manage input widgets, make the process of organizing a group of widgets simpler, or simply as a decorative widget for separating groups of widgets. Once a container is created, a layout manager still needs to be applied to the container widget itself.

Containers with Frames

`QFrame` widgets can enclose and group widgets as well as function as placeholders in windows. Using frames, you can set the appearance of other widgets to have raised, sunken, or flat appearances. The `QFrame` class is used as the base class for a few other container classes, including **`QToolBox`** and **`QStackedWidget`**. Table A-26 lists a few of the `QFrame` class's methods.

Table A-26 Select methods for QFrame

Method	Description
`setFrameRect(QRect)`	Sets the rectangle that the frame is drawn in
`setFrameShadow(shadow)`	Sets the frame's shadow, using flags such as `Plain`, `Raised`, or `Sunken`
`setFrameShape(shape)`	Sets the frame's shape, using flags such as `Box`, `Panel`, `HLine`, and `VLine`
`setLineWidth(int)`	Sets the width of line drawn around the frame

`QToolBox` widgets provide a series of pages or compartments in a column. To navigate through each of the pages, a tab is included at the top of each page. By clicking on the next tab, the user can view a new tab's contents. Some methods for QToolBox are listed in Table A-27.

Table A-27 A few of the QToolBox class's methods

Method	Description
`addItem(widget, text)`	Adds the `widget` in a new tab at the bottom of the `toolbox`
`insertItem(index, widget, text)`	Inserts the `widget` in a new tab at the given `index`
`indexOf(widget)`	Returns the index of the specified `widget`
`setCurrentIndex(index)`	Sets the `index` to a new item's index
`setCurrentWidget(widget)`	Makes the `widget` the current widget displayed in the toolbox

When the item in a `QToolBox` is changed, the `currentChanged(index)` signal is emitted.

The `QStackedWidget` has a similar function to `QToolBox`, displaying multiple widgets stacked on top of one another to conserve space in a window. However, there is a key difference: `QStackedWidget` does not provide a means for the user to switch between tabs. Therefore, other widgets, such as a `QComboBox` or a `QListWidget`, are used to navigate through the different pages.

The `QTabWidget` is another container class that is similar to `QStackedLayout` but provides the tabs necessary to switch pages.

Finally, `QGroupBox` widgets typically group together collections of radio buttons and checkboxes. The main visual difference from the `QFrame` class is the addition of a title.

QScrollArea

A scroll area can be added onto a child widget to display the contents within a frame. If the size of the frame changes, the scroll bars will appear, allowing the user to still view the entire child widget. A few class methods are listed in Table A-28. The manner in how the scroll bars appear can be controlled with the **`QAbstractScrollArea`** class's size policies.

Table A-28 Select methods for QScrollArea

Method	Description
`ensureVisible(x, y, xmargin, ymargin)`	Ensures the specified (`x`, `y`) coordinate with margins `xmargin` and `ymargin` remains visible in the viewport
`setAlignment(alignment)`	Sets the `alignment` of the scroll area's widget

Method	Description
`setWidget(widget)`	Sets the scroll area's `widget`
`setWidgetResizable(bool)`	If `False`, the scroll area abides by the child widget's size

QMdiArea

For **multiple-windowed GUIs** (**MDIs**) , the **`QMdiArea`** class provides the container for displaying multiple windows inside a single application window. **Subwindows** are instances of the **`QMdiSubWindow`** class and can be arranged in tiled or cascading patterns. The subwindows can work together, relaying information back and forth. A context menu could also be added to the MDI area widget as a means to conveniently switch between windows. Some methods for the MDI widget are found in Table A-29.

Table A-29 List of select QMdiArea methods

Method	Description
`addSubWindow(widget)`	Adds `widget` as a new subwindow to the MDI area
`activeSubWindow()`	Returns the active subwindow
`cascadeSubWindow()`	Arranges subwindows in a cascade pattern
`tileSubWindows()`	Arranges subwindows in a tiled pattern
`removeSubWindow(widget)`	Removes `widget` from the MDI area, where `widget` is a subwindow
`setBackground(background)`	Sets the QBrush `background` for the MDI area
`subWindowList(subwindows)`	Returns a list of `subwindows`
`setTabsClosable(bool)`	If `True`, close buttons are placed on each tab in the tabbed view
`setTabsMovable()`	If `True`, tabs within the tabbed view are movable

QtQuick and QML

As Qt and PyQt continue to evolve with each new version, more focus has gone into creating more dynamic and fluid user interfaces. This is especially true with Qt 6 and PyQt6.

With the `QtQuick` and `QtQml` modules, developers are able to use the Qt Modeling Language (QML) to build custom interfaces and components. `QtQuick` includes a number of classes for building a canvas for visualizing graphical components, handling user input, working with data, and handling graphical effects that are reminiscent of mobile applications.

Note that `QtQuick` is different from the `QtWidgets` API that we have used throughout most of this book. The QML syntax that `QtQuick` uses is based on embedded JavaScript. Using PyQt, we are able to create applications that connect to the QML code using Python. In many instances, you are even able to use classes such as `QtCore` and `QtGui` to communicate with the interface built using QML.

There are two links that may help you get started using `QtQuick`. The first is Qt's Qt Quick documentation at `https://doc.qt.io/qt-6/qtquick-index.html`. The second is the Riverbank documentation at `www.riverbankcomputing.com/static/Docs/PyQt6/qml.html#ref-integrating-qml`.

Qt Style Sheets

For a great reference of widgets and properties that can be manipulated with Qt Style Sheets, have a look at `https://doc.qt.io/qt-6/stylesheet-reference.html`.

Style sheets allow for customizing many aspects and behaviors of widgets. Table A-30 lists many of the properties that can be modified. Widgets support only certain properties, so be sure to check out Qt's documentation if you are not sure about which properties you can change.

Table A-30 List of properties that can be influenced using Qt Style Sheets

Property	**Description**

`alternate-background-color`	The alternate background color for `QAbstractItemView` widgets `QListView{`
	`alternate-background-color: blue; background: grey`
	`}`

`Background`	Shorthand for setting the background

`background-color`	Background color used for the widget
	`QPushButton{`
	`background-color: #49DE1F`
	`}`

Property	Description

`background-image`	The background image used for the widget
	`QFrame{`
	`background-image: url(images/black_cat.png)` }

`Border`	Shorthand for setting the widget’s border
	`QComboBox{`
	`border: 2px solid magenta`
	`}`

`border-top, border-right, border-bottom, border-left`	Shorthand for specifying sides of the widget’s border
`border-color`	The color for all sides of the widget’s border
`border-image`	Specifies an image to fill the border

`border-radius`	The radius of the border’s corners
	`QTextEdit{`
	`border-width: 1px;`
	`border-style: groove;`
	`border-radius: 3px`
	`}`

`border-style`	Specifies the style for all of the border’s edges
`border-width`	Specifies the width for all of the border’s edges
`color`	The color used for rendering text

`font`	Shorthand for defining a widget’s font
	`QRadioButton{`
	`font: bold italic large “Helvetica”`
	`}`

`font-family, font-size, font-style, font-weight`	Other properties used to individually set a font’s features
`height, width`	The height and width of a widget
`icon-size`	The width and height of a widget’s icon
`image`	The image drawn on a widget. Can use `url` or `svg`
`margin`	Specifies the widget’s margins. Just like `border`, specific sides can also be set

Property	**Description**
`max-height, max-width`	The widget’s maximum height or width
`min-height, min-width`	The widget’s minimum height or width
`outline`	The outline draws a widget’s border. Can also specify color, style, and radius
`padding`	Specifies the widget’s padding. Just like `border`, specific sides can also be set
`selection-color`	The foreground color of selected items to text
`spacing`	Sets the internal spacing in a widget

`text-align`	Specifies the alignment of text and icons inside of a widget
	`QPushButton{`
	`text-align: right`
	`}`

Qt Namespace

Throughout this book, you have come across numerous enums and flags that allow you to describe or modify the parameters, states, and appearances of widgets. The `Qt` class in the `QtCore` module organizes the multitude of identifiers in the Qt Namespace. A **namespace** in C++ is essentially used to organize the names of functions and variables into logical groups to prevent errors.

To get an idea of just how extensive Qt Namespace is, have a look at `https://doc.qt.io/qt-6/qt.html`. There you’ll find enums related to alignment, cursor style, date format, dock widget areas, keyboard buttons, window states, and more.

Summary

You have already used many of PyQt’s foundational classes for building graphical user interfaces while following along with this book. The Appendix provides references to help you analyze the programs found in this book and to learn more about the widgets, layouts, and style sheets used to design and build PyQt applications. The classes and methods contained here act as a guide to get you thinking about ways to build and improve your own programs.