J. M. Willman, *Beginning PyQt*
https://doi.org/10.1007/978-1-4842-7999-1_7

7. Handling Events in PyQt

Joshua M Willman[1]
(1) Sunnyvale, CA, USA

Since GUIs need to perform tasks, the widgets, windows, and other aspects of the application need to be able to react to the events that occur. Whether caused by the user or by the underlying system, the events, and possibly data, need to be delivered to their appropriate locations and handled accordingly.

In this chapter, you will

- Find out more about signals, slots, and event handling
- Learn how to modify key press and mouse and enter event handlers
- Explore how to create custom signals using `pyqtSignal`

This chapter is all about handling events and modifying the behaviors of the built-in functions in PyQt.

Event Handling in PyQt

Events in Qt are objects created from the **`QEvent`** class . The event objects describe different types of interactions that can occur in a GUI as a result of what happens, either caused by a user or by some kind of system activity outside of the application. These events begin once the application's main event loop starts.

Most events, whether the press of a key, click of a mouse, resizing of a window, or dragging and dropping of a widget or data, have their own subclass of `QEvent` that generates an event object and passes it on to the appropriate `QObject` by calling the event() method, which in turn is handled by the suitable event handler. (Recall that `QWidget` inherits `QObject`.) The response from the event is used to determine whether it was accepted or disregarded.

More information about event handling can be found at https://doc.qt.io/qt-6/eventsandfilters.html.

Let's take a look at signals and slots and event handlers in the following subsections and think about their purposes and their differences.

Using Signals and Slots

The concept of signals and slots in PyQt was briefly introduced in Chapter 3. Widgets in PyQt use signals and slots to communicate between objects. Just like events, signals can be generated by a user's actions or by the internal system. Slots are methods that are executed in response to the signal. For example, when a `QPushButton` is pressed, it emits a `clicked` signal. This signal could be connected to a built-in PyQt slot, such as `close()` to allow a user to quit an application, or to a custom-made slot, which is typically a Python function. Signals are also useful be-

cause they can be used to send additional data to a slot and provide more information about an event.

The `clicked` signal is but one of many predefined Qt signals. The type of signals that can be emitted differs according to the widget class. PyQt delivers events to widgets by calling specific, predefined event handling functions. These can range from functions related to window operations such as `show()` or `close()`, to GUI appearances with `setStyleSheet()`, to mouse press and release events, and more.

Have a look at www.riverbankcomputing.com/static/Docs/PyQt6/signals_slots.html for more information about signals and slots in PyQt6.

Using Event Handlers to Handle Events

Event handlers are the functions that respond to an event. While a `QEvent` subclass is created to deliver the event, a corresponding `QWidget` method will actually handle the event. If you remember in Chapter 3, the closeEvent() event handler was used to close windows. The class that creates the close event object is **`QCloseEvent`**.

> **Note** You may not always be able to handle all of the functionality in an event handler that you modify. When this is the case, you can use an `if-else` statement. In the `if` condition, specify how to react to the event, and in the `else` clause, call the base class's implementation. So for `QCloseEvent`, you would include `super().closeEvent(event)` in the `else` clause. This portion will take care of any default behaviors you did not implement or may have missed.

Difference Between Signals and Slots and Event Handlers

While there is some overlap between the two, signals and slots are typically used for communication between the different widgets and other PyQt classes. Events are generated by an outside activity and delivered through the event loop by `QApplication`.

Another important difference is that you are notified when a signal is emitted and take action accordingly. Events need to be handled whenever they occur.

Finally, we can use signals with widgets to improve their capabilities, but you will need to reimplement event handlers when modifying a widget's functionalities.

In many cases, you will use signals and slots and event handlers together to complete tasks.

The following section shows a simple example of how to reimplement the `keyPressEvent()` function.

Handling Key Events

When keys are pressed or released, a **`QKeyEvent`** is created. Key events are sent to the widget that currently has keyboard focus. We can then reimplement the following `QWidget` key event handlers to deal with the event:

- `keyPressEvent()` – Handles a key event when the key is pressed
- `keyReleaseEvent()` – Handles a key event when the key is released

Figure 7-1 shows the GUI that we'll code, which demonstrates how to modify `keyPressEvent()`.

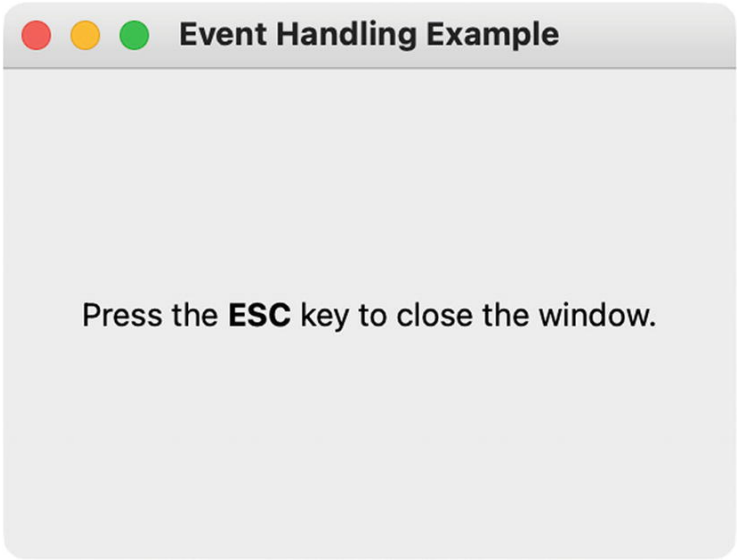

Figure 7-1 A window that closes when the user presses the Escape key

Some key names include `Key_Escape`, `Key_Return`, `Key_Up`, `Key_Down`, `Key_Space`, `Key_0`, `Key_1`, and so on. A full list of `Qt.Key` enum keyboard codes can be found at https://doc.qt.io/qt-6/qt.html#Key-enum.

Explanation for Handling Key Events

For Listing 7-1, we'll create a simple `MainWindow` class that inherits `QMainWindow`. The imports for this application are also fairly simple. The `MainWindow` class inherits `QMainWindow` so that we don't have to import any layout managers for the single `QLabel` object.

```
# key_events.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import QApplication,
QMainWindow, QLabel
from PyQt6.QtCore import Qt
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initializeUI()
    def initializeUI(self):
        """Set up the application's GUI."""
        self.setGeometry(100, 100, 300, 200)
        self.setWindowTitle("Event Handling Example")
        info_label = QLabel(
            """<p align='center'>Press the <b>ESC</b>
key
            to close the window.</p>""")
        self.setCentralWidget(info_label)
        self.show()
    def keyPressEvent(self, event):
        """Reimplement the key press event to close
the
        window."""
        if event.key() == Qt.Key.Key_Escape:
            print("Application closed.")
            self.close()
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec())
```

Listing 7-1 Code to demonstrate how to modify key event handlers

Whenever a user presses a key on the keyboard, it sends a signal to the computer. If you want to give certain keys abilities, then you will need to use the `keyPressEvent()`.

The `keyPressEvent()` function checks for events, which in this case are the signals being sent from keys. If the key pressed is the `Escape` key, then the application calls the `close()` function to quit the application. Different keys can be accessed using `Qt.Key`, and you can use those different keys to perform any number of actions.

Handling Mouse Events

Mouse events are handled by the **`QMouseEvent`** class. For mouse events, we need to be able to find out when a mouse button is pressed, released, and double-clicked and when the mouse moves while clicked. There is also an event class, **`QEnterEvent`**, that is useful for finding out if the mouse has entered or left the window or a particular widget. Enter events are also useful for collecting information about the mouse cursor's position.

The `QWidget` mouse event handlers we'll be using include the following:

- `mousePressEvent()` – Handles events when the mouse button is pressed.
- `mouseReleaseEvent()` – Handles events when the mouse button is released.
- `mouseMoveEvent()` – Handles events when the mouse button is pressed and moved. Turn on mouse tracking to enable move events even if a mouse button is not pressed with `QWidget.setMouseTracking(True)`.
- `mouseDoubleClickEvent()` – Handles events when the mouse button is double-clicked.

For the enter events, we'll use the following event handlers:

- `enterEvent()` – Handles when the mouse cursor enters a widget
- `leaveEvent()` – Handles when the mouse cursor leaves a widget

For the GUI in Figure 7-2, there is only the image in the left window without any textual information when the program first starts. When a mouse enters the main window, the image in the window will change to what is shown in the right screenshot in Figure 7-2. If the user clicks or releases the mouse button, a label in the widget will update to let them know which mouse button, left or right, was used. Double-clicking in the window will change the image. Lastly, the x and y coordinates of the mouse's position are displayed on the screen when the mouse is pressed and moving.

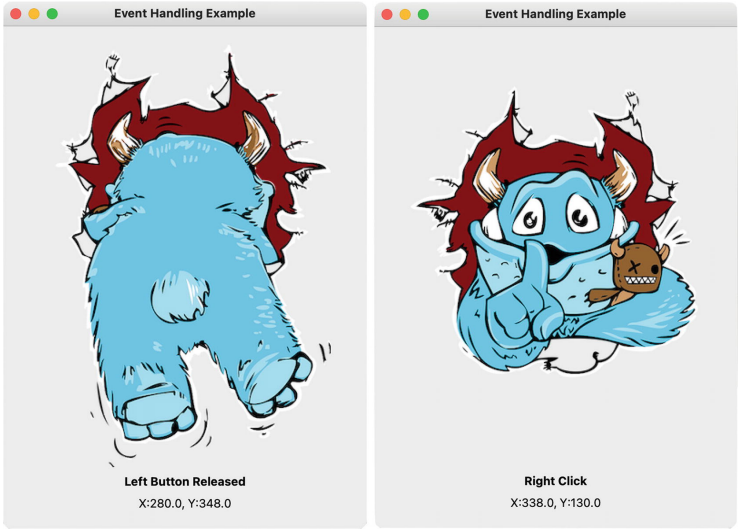

Figure 7-2 The images and information in the window change based on the mouse events. Images from https://pixabay.com

Be sure to download the `images` folder from the GitHub repository for this example.

Explanation for Handling Mouse Events

For this example, we can use the `basic_window.py` script from Chapter 1. In Listing 7-2, let's set up the main window and the `setUpMainWindow()` method. The window consists of three `QLabel` objects, one for displaying images and the other two for relaying information about the mouse events to the user.

```
# mouse_events.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication, QWidget, QLabel,
    QVBoxLayout)
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QPixmap
class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
        self.initializeUI()
    def initializeUI(self):
        """Set up the application's GUI."""
        self.setMinimumSize(400, 300)
        self.setWindowTitle("Event Handling Example")
        self.setUpMainWindow()
        self.show()
    def setUpMainWindow(self):
        self.image_label = QLabel()
        self.image_label.setPixmap(QPixmap("images/back.png"))
        self.image_label.setAlignment(
            Qt.AlignmentFlag.AlignCenter)
        self.info_label = QLabel("")
        self.info_label.setAlignment(
            Qt.AlignmentFlag.AlignCenter)
        self.pos_label = QLabel("")
        self.pos_label.setAlignment(
            Qt.AlignmentFlag.AlignCenter)
        main_h_box = QVBoxLayout()
        main_h_box.addStretch()
        main_h_box.addWidget(self.image_label)
        main_h_box.addStretch()
        main_h_box.addWidget(self.info_label)
```



```
        main_h_box.addWidget(self.pos_label)
        self.setLayout(main_h_box)
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec())
```

Listing 7-2 Code for setting up the main window in the modifying mouse event handlers example

The `addStretch()` method is used before and after `image_label` in `main_h_box` to make sure the images stay centered in the window.

Whenever the mouse cursor enters a window, `image_label` will display a different image. To change the image back, we can use `leaveEvent()` to check when the mouse has left the widget. This is done in Listing 7-3.

```
# mouse_events.py
    def enterEvent(self, event):
        self.image_label.setPixmap(
            QPixmap("images/front.png"))
    def leaveEvent(self, event):
        self.image_label.setPixmap(QPixmap("images/back.png"))
```

Listing 7-3 Code for the enterEvent() and leaveEvent() event handlers

In PyQt6, `QMouseEvent` inherits a few methods from **`QPointerEvent`** that can provide more information about which mouse buttons are clicked or where the mouse is in the window or on the computer screen. These include the following:

- `button()` – Returns which button caused the event.
- `buttons()` – Returns the state of the buttons, giving access to which combination of buttons caused the event using an OR operator.
- `globalPosition()` – Returns the point coordinates of the event on the computer screen.
- `position()` – Returns the current point coordinates of the mouse relative to the widget that caused the event. The val-

ues returned refer to points within the window or widget.

Both `globalPosition()` and `position()` have `x()` and `y()` methods for collecting horizontal or vertical values. We'll use a few of these methods in Listing 7-4.

```
# mouse_events.py
    def mouseMoveEvent(self, event):
        """Print the mouse position while clicked and
        moving."""
        if self.underMouse():
            self.pos_label.setText(
                f"""<p>X:{event.position().x()},
                    Y:{event.position().y()}</p>""")
    def mousePressEvent(self, event):
        """Determine which button was clicked."""
        if event.button() == Qt.MouseButton.LeftButton:
            self.info_label.setText("<b>Left Click</b>")
        if event.button() == Qt.MouseButton.RightButton:
            self.info_label.setText("<b>Right Click</b>")
    def mouseReleaseEvent(self, event):
        """Determine which button was released."""
        if event.button() == Qt.MouseButton.LeftButton:
            self.info_label.setText(
                "<b>Left Button Released</b>")
        if event.button() == Qt.MouseButton.RightButton:
            self.info_label.setText(
                "<b>Right Button Released</b>")
    def mouseDoubleClickEvent(self, event):
        self.image_label.setPixmap(QPixmap("images/boom.png"))
```

Listing 7-4 Code that demonstrates how to modify mouse event handlers

The mouse's x and y values are displayed in `pos_label` using `position()` in `mouseMoveEvent()`. For `mousePressEvent()`, we'll simply update the text of `info_label` depending upon which mouse button is clicked. The `mouseReleaseEvent()` will do something similar, but when the button is released. For `mouseDoubleClickEvent()`, `pixmap` is updated to look like Figure 7-3. Moving the mouse out of the window causes `leaveEvent()` to be called, showing the images in Figure 7-2 again.

Figure 7-3 The image in the screen changes when the mouse is double-clicked

After seeing how to modify event handlers, now is a good time to learn how to create your own signals.

Creating Custom Signals

We have taken a look at some of PyQt's predefined signals and slots in previous chapters. For many of those applications, we have also seen how to create custom slots to handle the signals emitted from widgets. The custom slots were simply Python functions or methods.

Now let's see how we can create custom signals using `pyqtSignal` to change a widget's style sheet. Using **`pyqtSignal`**, new signals can be defined for a class. Just like predefined signals, you can also pass types of information, such as Python strings, integers, dictionaries, or lists, as arguments to the `pyqtSignal` you create.

For the GUI in Figure 7-4, a user can change the background color of the lower `QLabel` widget by pressing the up or down arrow keys on their keyboard. A **closed** signal, one that takes no arguments, will be emitted when a key is pressed.

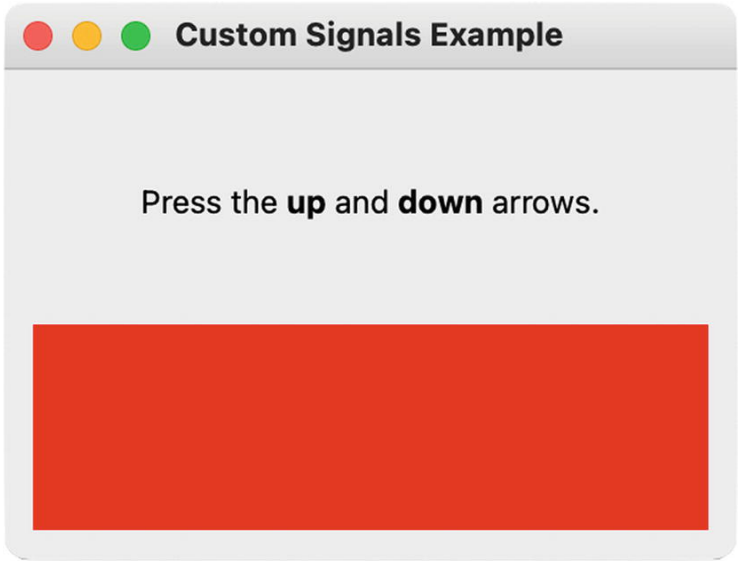

Figure 7-4 The color of the label will change when the up and down arrows are pressed

Explanation for Creating Custom Signals

This example creates a simple GUI with a `QLabel` object as the central widget of the main window. The `pyqtSignal` factory and `QObject` classes are imported from the `QtCore` module. The `QtCore` module and `QObject` class provide the mechanics for signals and slots.

Before creating the `MainWindow` class in Listing 7-5, we'll first create a class, `SendSignal`, that inherits `QObject`.

```
# custom_signal.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication,
QMainWindow,
    QWidget, QLabel, QVBoxLayout)
from PyQt6.QtCore import Qt, pyqtSignal, QObject
class SendSignal(QObject):
    """Define a signal, change_style, that takes no
    arguments."""
```

```
    change_style = pyqtSignal()
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initializeUI()
    def initializeUI(self):
        """Set up the application's GUI."""
        self.setGeometry(100, 100, 300, 200)
        self.setWindowTitle("Create Custom Signals")
        self.setUpMainWindow()
        self.show()
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec())
```

Listing 7-5 Creating a custom signal to change the background color of a QLabel widget

The `SendSignal` class creates a new signal called `change_style` from the `pyqtSignal` factory. To use this signal, we'll first need to create an instance of `SendSignal`, simply called `sig`, in Listing 7-6. To use the custom signal you created, call the `change_style` instance from `sig`, and use `connect()` to connect the signal to a slot, in this case, `changeBackground()`.

```
# custom_signal.py
    def setUpMainWindow(self):
        """Create and arrange widgets in the main window."""
        self.index = 0 # Index of items in list
        self.direction = ""
        # Create instance of SendSignal class, and
        # connect change_style signal to a slot
        self.sig = SendSignal()
        self.sig.change_style.connect(self.changeBackground)
        header_label = QLabel(
            """<p align='center'>Press the <b>up</b> and
            <b>down</b> arrows.</p>""")
        self.colors_list = ["red", "orange", "yellow",
                            "green", "blue", "purple"]
        self.label = QLabel()
        self.label.setStyleSheet(f"""background-color:
            {self.colors_list[self.index]}""")
        main_v_box = QVBoxLayout()
        main_v_box.addWidget(header_label)
        main_v_box.addWidget(self.label)
```



```
        container = QWidget()
        container.setLayout(main_v_box)
        self.setCentralWidget(container)
```

Listing 7-6 Code for the setUpMainWindow() method

The rest of `setUpMainWindow()` instantiates the two `QLabel` widgets and creates a list of colors that are used by `label` to specify the background in its style sheet.

This signal will be emitted whenever the user presses either the up arrow key or the down arrow key in `keyPressEvent()`.

When the user presses `Key_Up`, `direction` is set equal to `"up"`, and a `change_style` signal is emitted. To emit a custom signal, you'll need to call `emit()` at the point in your application where the signal needs to be triggered. An example for `sig` is shown in the following line:

```
self.sig.change_style.emit()
```

This signal is connected to the `changeBackground()` slot that updates the color of the label by checking the index of `colors_list` and updating the color using `setStyleSheet()` in Listing 7-7.

```
# custom_signal.py
    def keyPressEvent(self, event):
        """Reimplement how the key press event is
handled."""
        if event.key() == Qt.Key.Key_Up:
            self.direction = "up"
            self.sig.change_style.emit()
        elif event.key() == Qt.Key.Key_Down:
            self.direction = "down"
            self.sig.change_style.emit()
    def changeBackground(self):
        """Change the background of the label widget
when
        a keyPressEvent signal is emitted."""
        if self.direction == "up" and \
            self.index < len(self.colors_list) - 1:
            self.index = self.index + 1
            self.label.setStyleSheet(f"""background-
color:
            {self.colors_list[self.index]}""")
```

```
        elif self.direction == "down" and self.index
> 0:
            self.index = self.index - 1
            self.label.setStyleSheet(f"""background-
color:
            {self.colors_list[self.index]}""")
```

Listing 7-7 Code for handling keyPressEvent() and the slot for changing the background color

It works in a similar fashion when the down key is pressed. Remember that custom signals can take data types as arguments, so don't worry if you need to pass along information to your other widgets or classes.

Summary

Handling events is a critical component of GUI development. With PyQt, this can be accomplished either through signals and slots or by the event classes and their corresponding event handlers. Either way, you may often find yourself extending the abilities of a widget class by creating custom signals using `pyqtSignal` or reimplementing the base functionality provided by Qt's various event handler methods.

We took a look at both these concepts in this chapter, changing the behaviors of key press and mouse event handlers and creating a custom signal to modify the appearance of a label.

In the next chapter, we'll take a look at using the application Qt Designer to create PyQt applications and simplifying the process for arranging widgets in a GUI window.