setting your widget's position. Another reason has to deal with resizing a window. If you were to adjust the size of the window by dragging on the bottom-right corner, you'll notice that the widgets don't move or stretch. Qt's layout classes are great for handling this and other issues. We'll discuss using the layout classes in Chapter 4.

You might think learning using `move()` is a waste of time, but it can be very useful to understand how to use pixel values to manipulate widgets, especially when we begin dealing with more advanced topics like animations and graphics classes.

The image is loaded in a similar fashion, creating a `world_label` object to be placed in the main window. Then we construct a `QPixmap` of the image and use `setPixmap()` to set the image displayed onto the `world_label`. The image's absolute location is set using `move()`. An exception is thrown if the image cannot be found.

Each of PyQt's different classes has their own methods that can be used to customize and change their look and functionality. In the Appendix, you can find a list of the widgets used in this book along with some of the more common methods you are likely to use to modify them.

Once you run the program, you should see a window like Figure 2-1 appear on your screen. In the next section, you'll build a slightly more complex GUI using `QLabel` widgets.

Project 2.1 – User Profile GUI

A user profile is used to visually display personal data. The data in the profile helps to associate certain characteristics with that user and assists others in learning more about that individual. Depending upon the goal of the application, the information and appearance of the profile will change.

User profiles like the one displayed in Figure 2-2 often have a number of parameters that are either mandatory or optional and allow for some level of customization to fit the preferences of the user, such as a profile image or background colors. Many of them contain similar features, such as the user's name or an "About" section.

Figure 2-2. *The User Profile GUI that displays a user's information*

In the next section, we'll break apart Figure 2-2 and think about how the label widgets will be arranged in the window.

Designing the User Profile GUI

Typical user profile applications often use a combination of different elements, both interactive and static. The schematic in Figure 2-3 focuses on utilizing solely static `QLabel` widgets for displaying information in the window.

If you compare Figure 2-3 with Figure 2-2, you will notice the similarity with how they are arranged. The user interface can be divided into two parts. The upper portion uses `QLabel` objects that display a profile image that lies on top of a background image.

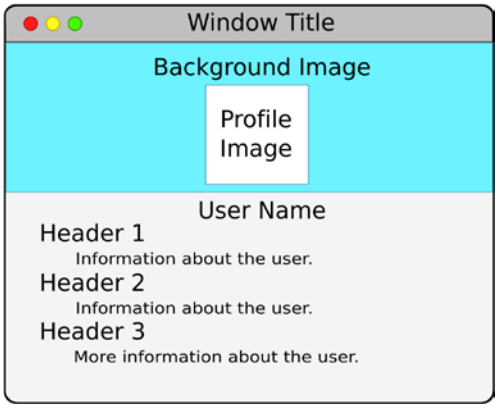

Figure 2-3. *Schematic for the User Profile GUI*

The bottom portion shows the user's information with multiple `QLabel` widgets, with the textual information arranged vertically and broken down into smaller sections, delineated by the use of different font sizes.

Explanation for the User Profile GUI

Similar to the last application, we'll begin by using the template GUI from Chapter 1 as the foundation for the User Profile's main window in Listing 2-3.

Listing 2-3. Code for setting up the User Profile GUI's main window

```
# user_profile.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import QApplication, QWidget, QLabel
from PyQt6.QtGui import QFont, QPixmap
class MainWindow(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()
```

```
    def initializeUI(self):
        """Set up the application's GUI."""
        self.setGeometry(50, 50, 250, 400)
        self.setWindowTitle("2.1 - User Profile GUI")

        self.setUpMainWindow()
        self.show()

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec())
```

For the User Profile GUI, import the same classes and modules as the earlier application along with the addition of one new class, the `QFont` class from the `QtGui` module, which allows for us to modify the size and types of fonts in our application. This is perfect for creating the different header sizes.

Before creating `setUpMainWindow()`, let's create a separate method in `MainWindow`, seen in Listing 2-4, that will handle loading the different images and creating `QLabel` objects to display them.

Listing 2-4. Code for createImageLabels() in the User Profile GUI

```
# user_profile.py
    def createImageLabels(self):
        """Open image files and create image labels."""
        images = ["images/skyblue.png",
                  "images/profile_image.png"]

        for image in images:
            try:
                with open(image):
                    label = QLabel(self)
                    pixmap = QPixmap(image)
                    label.setPixmap(pixmap)
```

```
                if image == "images/profile_image.png":
                    label.move(80, 20)
            except FileNotFoundError as error:
                print(f"Image not found.\nError: {error}")
```

The `images` list contains the specific file locations that will be used for both the blue background and the user's profile image in the top part of the window. Using a `for` loop, iterate through the list's items, create a `QLabel` object for each, instantiate a `QPixmap` object, set the `pixmap` for the `label`, and if the `image` is the profile image, center it in the window using `move()`. Using `move()` and absolute positioning, you can easily overlap images, but you will need to load the images in order from the bottom-most image to the top-most.

We can now create the `MainWindow` method `setUpMainWindow()` in Listing 2-5 where `createImageLabels()` will be called.

Listing 2-5. Code for the User Profile GUI's setUpMainWindow() method

```
# user_profile.py
    def setUpMainWindow(self):
        """Create the labels to be displayed in the window."""
        self.createImageLabels()

        user_label = QLabel(self)
        user_label.setText("John Doe")
        user_label.setFont(QFont("Arial", 20))
        user_label.move(85, 140)

        bio_label = QLabel(self)
        bio_label.setText("Biography")
        bio_label.setFont(QFont("Arial", 17))
        bio_label.move(15, 170)

        about_label = QLabel(self)
        about_label.setText("I'm a Software Engineer with 10 years\
            experience creating awesome code.")
        about_label.setWordWrap(True)
        about_label.move(15, 190)
```

After the image labels are created, several QLabel objects for showing text are instantiated. For example, the user_label displays the user's name using setText() in the window. You can set a QLabel widget's font with the method setFont(). Be sure to pass a QFont object and specify the type of font and its size. The user_label is then centered in the window using move(). Other labels are created in a similar manner.

Listing 2-6 continues to create and arrange QLabel widgets in the main window.

Listing 2-6. Arranging more labels in the setUpMainWindow() method

```
# user_profile.py
        skills_label = QLabel(self)
        skills_label.setText("Skills")
        skills_label.setFont(QFont("Arial", 17))
        skills_label.move(15, 240)

        languages_label = QLabel(self)
        languages_label.setText("Python  |  PHP  |  SQL  |  JavaScript")
        languages_label.move(15, 260)
```

More labels are created. Notice how the x value in move() stays at 15, leaving a small space on the left side of the window, and the y value gradually increases, placing each subsequent label lower. More labels are added to the GUI in Listing 2-7.

Listing 2-7. Arranging even more labels in the setUpMainWindow() method

```
# user_profile.py
        experience_label = QLabel(self)
        experience_label.setText("Experience")
        experience_label.setFont(QFont("Arial", 17))
        experience_label.move(15, 290)

        developer_label = QLabel(self)
        developer_label.setText("Python Developer")
        developer_label.move(15, 310)

        dev_dates_label = QLabel(self)
        dev_dates_label.setText("Mar 2011 - Present")
        dev_dates_label.setFont(QFont("Arial", 10))
        dev_dates_label.move(15, 330)
```

```
        driver_label = QLabel(self)
        driver_label.setText("Pizza Delivery Driver")
        driver_label.move(15, 350)

        driver_dates_label = QLabel(self)
        driver_dates_label.setText("Aug 2015 - Dec 2017")
        driver_dates_label.setFont(QFont("Arial", 10))
        driver_dates_label.move(15, 370)
```

Running the application now, you will see a window appear like the one in Figure 2-2.

Summary

In this chapter, we discovered how to add and arrange widgets in a GUI window. The `QLabel` widget is a fundamental class and is not only great for displaying text but can also be used with other PyQt classes, such as `QPixmap` for displaying images or `QFont` for changing the label's text style or size. Each one of the PyQt classes includes various methods for extending their capabilities and appearance. Examples of those can be found in the Appendix.

In the next chapter, we'll explore a number of different widget classes, including `QPushButton` and `QLineEdit`, that will allow users to interact with the applications that you develop.

CHAPTER 3

Adding More Functionality with Widgets

What good is a user interface if it isn't interactive? This chapter is all about learning how to use widgets to make responsive user interfaces that react to a user's interaction, handle different events, and relay important information back to the user. We will take a look at a few common widgets and see how to use them to design and build GUI applications.

In this chapter, you will

- Be introduced to event handling and Qt's signals and slots mechanism
- Build GUIs using new widget classes, including `QPushButton`, `QLineEdit`, `QCheckBox`, and `QMessageBox`
- Learn about useful methods for aligning text and adjusting widget sizes
- Discover more about windows and dialog boxes and see how to create classes that inherit from `QDialog`
- Create an application that teaches how to handle multiple windows

Before jumping into any code, let's learn a little about event handling in PyQt.

Event Handlers and Signals and Slots

GUIs are **event driven**, meaning that they respond to events that are created by the user, from a keyboard or a mouse, or by events caused by the system, such as a timer or when connecting to Bluetooth. In Qt, special kinds of events are even generated to handle

J. M. Willman, *Beginning PyQt*, https://doi.org/10.1007/978-1-4842-7999-1_3

communication between widgets. No matter how they are generated, the application needs to listen for those events and respond to them appropriately. This is known as **event handling**. When `exec()` is called, the application begins listening for events until the program is closed.

In PyQt, event handling is handled in one of two ways – either through event handlers or with signals and slots. **Event handlers** take care of events. There are different types of events that can be handled, such as `paintEvent()` for repainting the look of a widget or `keyPressEvent()` that handles key presses. In Qt, events are objects created from the `QEvent` class.

The communication between objects in Qt, such as widgets, is handled by signals and slots. **Signals** are generated whenever an event occurs, such as when a button is clicked or a checkbox is toggled on or off. Those signals then need to be handled in some way. **Slots** are the methods that are connected to an event and executed in response to the signal. Slots can either be built-in PyQt functions or Python functions that you create yourself.

Each PyQt class has its own assortment of signals, and many of them are inherited from parent classes. Let's look at an example. Whenever a user clicks a button in the window, that button click will send out, or **emit**, a signal:

```
button.clicked.connect(self.buttonClicked)
```

Here, `button` is a widget, and `clicked` is the signal. In order to make use of that signal, we must use `connect()` to call some function, which in this case is `buttonClicked()`, which is the slot. The `buttonClicked()` method could then perform some action, such as opening a new window. Many signals also pass along additional information to the slot, such as a Boolean value that tells whether or not the button was pressed.

Signals and slots, and even making custom signals, will be covered in Chapter 7. For now, let's take a look at a widget that is perfect for demonstrating signals and slots.

The QPushButton Widget

The **`QPushButton`** widget can be used to perform actions and make choices. When you click on the `QPushButton` widget, it sends out a signal that can be connected to a function. While you might typically encounter buttons with text that say OK, Next, Cancel, Close, Yes, or No, you can also create your own buttons with descriptive text or icons.

Note There are different kinds of button classes with different usages, such as `QToolButton` for selecting items in toolbars and `QRadioButton` for creating groups of buttons where only a single selection can be made.

In this first example, you are going to set up a `QPushButton` that, when clicked, uses signals and slots to change the text of a `QLabel` widget and shows how to handle closing an application's main window.

Let's take a look at how to build the GUI.

Explanation for Using QPushButton

Open a new file and copy the code from the empty window script from Chapter 1 into it. As you can see in Listing 3-1, you'll need to import a few more classes, including the `QPushButton` class from `QtWidgets`. The `QtCore` module contains a bunch of non-GUI-related classes. The `Qt` class refers to the **Qt Namespace**, which contains many identifiers used for setting the properties of widgets and other classes.

Listing 3-1. Setting up the main window for using QPushButton widgets

```
# buttons.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication, QWidget, QLabel,
    QPushButton)
from PyQt6.QtCore import Qt

class MainWindow(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()
```

```
    def initializeUI(self):
        """Set up the application's GUI."""
        self.setGeometry(100, 100, 250, 150)
        self.setWindowTitle("QPushButton Example")

        self.setUpMainWindow()
        self.show()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec())
```

Be sure to set the main window's starting x and y positions and size using `setGeometry()`. Then set the window's title and call the `setUpMainWindow()` method which we'll create in Listing 3-2.

Listing 3-2. The setUpMainWindow() method for using buttons

```
# buttons.py
    def setUpMainWindow(self):
        """Create and arrange widgets in the main window."""
        self.times_pressed = 0

        self.name_label = QLabel(
            "Don't push the button.", self)

        self.name_label.setAlignment(
            Qt.AlignmentFlag.AlignCenter)
        self.name_label.move(60, 30)

        self.button = QPushButton("Push Me", self)
        self.button.move(80, 70)
        self.button.clicked.connect(self.buttonClicked)
```

The variable `times_pressed` will be used to keep track of how many times `button` is pressed. The window for this application only contains a `QLabel` and a `QPushButton`. Rather than using `setText()` to assign the text for `name_label`, we can instead pass the text we want to display as the first argument when instantiating the `QLabel` object.

It is possible to align the contents of widgets that display text. To do so, use `setAlignment()`, and because we're using PyQt6, be sure to pass the full enum type, `Qt.AlignmentFlag`. There are different kinds of alignment flags, some of which are

- `AlignLeft` – Aligns text to the left edge
- `AlignRight` – Aligns text to the right edge
- `AlignHCenter` and `AlignVCenter` – Centers text horizontally and vertically, respectively
- `AlignTop` and `AlignBottom` – Aligns text to the top and bottom, respectively

Here, let's use `AlignCenter`, which is a combination of `AlignVCenter` and `AlignHCenter`. Use `move()` to set the absolute position of the widget.

Note Instead of using setters, many of the properties for widgets can be set by passing them as arguments to a widget instance. For example, rather than using `setAlignment()`, you could set the alignment for the label by passing the keyword argument `alignment=Qt.AlignmentFlag.AlignCenter` after `self`.

Next, create the `QPushButton` object, and pass the button's text and `self`, a reference to the `MainWindow` class, as arguments. Clicking on the button will emit the `clicked` signal, which is connected to the `buttonClicked()` slot (shown in Listing 3-3).

Listing 3-3. Code for the buttonClicked() slot

```
# buttons.py
    def buttonClicked(self):
        """Handle when the button is clicked.
        Demonstrates how to change text for widgets,
        update their sizes and locations, and how to
        close the window due to events."""
        self.times_pressed += 1

        if self.times_pressed == 1:
            self.name_label.setText("Why'd you press me?")
        if self.times_pressed == 2:
```

```
            self.name_label.setText("I'm warning you.")
            self.button.setText("Feelin' Lucky?")
            self.button.adjustSize()
            self.button.move(70, 70)
        if self.times_pressed == 3:
            print("The window has been closed.")
            self.close()
```

In `buttonClicked()`, we'll first update the variable `times_pressed`. Next, there are a series of `if` statements that depend upon the value of `times_pressed`. You can update text values for widgets even after they have been created. If `times_pressed` equals `1`, change the text for `name_label` using `setText()`.

For a value of `2`, change the text for both `name_label` and `button`. For `button`, you will also need to adjust its size to fit the longer text value. Since `QPushButton` inherits `QWidget`, we can use the `QWidget` method `adjustSize()` to change the size of `name_label` in order to fit the longer text. Since absolute positioning is being used to arrange widgets, you'll also need to use `move()` to center `button` in the window. You can see examples of the text changing in Figure 3-1.

Figure 3-1. *Clicking on the QPushButton will change the label's text and, eventually, the button's text*

Finally, for `3`, the `QWidget` method `close()` is used to close widgets. In this case, `self.close()` is referring to the main window and closes the application. We'll look more at closing events later in the "Project 3.1 – Login GUI and Registration Dialog" section.

Next, we'll look at a widget that is useful for collecting user input.

The QLineEdit Widget

It is often necessary for a user to input a single line of text, such as a username or a password. With the **`QLineEdit`** widget, you can collect data from someone. `QLineEdit` also supports normal text editing functions such as cut, copy, and paste, and redo or undo. There are also additional capabilities for hiding text when it is entered, using placeholder text, or even setting a limit on the length of the text.

Tip If you need multiple lines for a user to enter text, use the `QTextEdit` widget instead.

The GUI you will build in Figure 3-2 demonstrates how to set up and use `QLineEdit` widgets. You can use other widgets, such as `QPushButton`, along with signals and slots to retrieve the text in a `QLineEdit` object or clear its text.

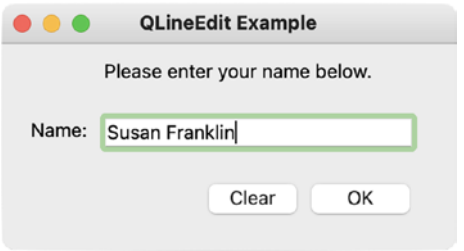

Figure 3-2. *QLineEdit and QPushButton widgets used for collecting and clearing text*

In the next section, you'll find out how to use `QLineEdit`.

Explanation for Using QLineEdit

Listing 3-4 sets up the main window seen in Figure 3-2. You'll need to import different widget classes, including `QLabel`, `QLineEdit`, and `QPushButton`, as well as `Qt` from the `QtCore` module into the empty window script from Chapter 1.

Listing 3-4. Setting up the main window for using QLineEdit widgets

```
# line_edits.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication, QWidget,
    QLabel, QLineEdit, QPushButton)
from PyQt6.QtCore import Qt

class MainWindow(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """Set up the application's GUI."""
        self.setMaximumSize(310, 130)
        self.setWindowTitle("QLineEdit Example")

        self.setUpMainWindow()
        self.show()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec())
```

Previous examples have used `setGeometry()` for setting the location and size of a window on the screen. One thing you can do is restrict the size of the window. Here, let's use the `QWidget` method `setMaximumSize()` and pass the maximum width and height for `MainWindow`. Some other methods for setting window sizes include the following:

- `setMinimumSize()` – Sets the widget's minimum size
- `setMinimumHeight()` and `setMinimumWidth()` – Sets the widget's minimum height and width, respectively
- `setMaximumHeight()` and `setMaximumWidth()` – Sets the widget's maximum height and width, respectively
- `setFixedSize()` – Sets the maximum and minimum sizes for the widget, preventing it from changing sizes