PyQt Comprehensive Guide[1]

CS355/CE373 Database Systems
Fall 2024

Dhanani School of Science and Engineering

Habib University

[1]by CS Research Assistants - Shafaq Fatima Mughal & Shayan Aamir

Contents

This guide will assist you in learning how to use PyQt6 to create desktop applications with Python. It provides an easy and comprehensive understanding of the subject that will be helpful in your labs. It will even eventually help with the final project of this course.

Note: This guide has been adopted from the PyQt6 documentation but serves in no way as a replacement. If you want to learn more about PyQt6 and its capabilities, we recommend checking out the official PyQt6 documentation.

1 Installation

PyQt6 for Windows can be installed as for any other application or library. As of Qt 5.6 installers are available to install via PyPi, the Python Package archive. To install PyQt6 from Python simply run:

```
pip install pyqt6
```

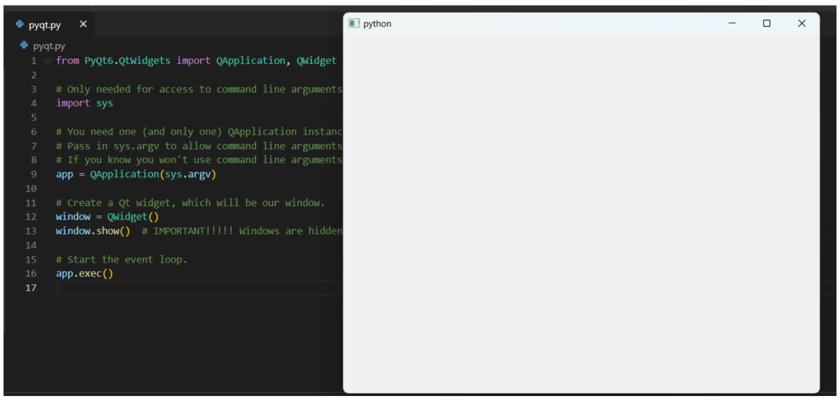
After the installation is finished, you should be able to successfully import PyQt6 in your Python files.

2 Creating a PyQt6 App

To start, create a new Python file, you can call it whatever you like (e.g. `app.py`) and save it somewhere accessible. We'll write our simple app in this file:

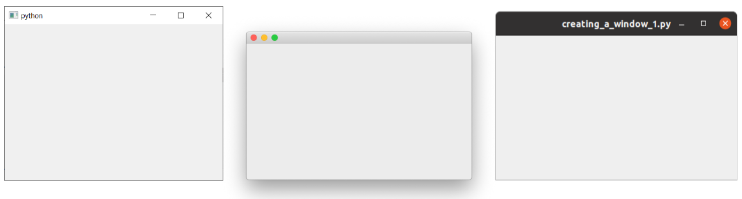
```python
from PyQt6.QtWidgets import QApplication, QWidget

# Only needed for access to command line arguments
import sys

# You need one (and only one) QApplication instance per application.
# Pass in sys.argv to allow command line arguments for your app.
# If you know you won't use command line arguments QApplication([]) works too.
app = QApplication(sys.argv)

# Create a Qt widget, which will be our window.
window = QWidget()
window.show()  # IMPORTANT!!!!! Windows are hidden by default.

# Start the event loop.
app.exec()

# Your application won't reach here until you exit and the event
# loop has stopped.
```

Running this code will give you an empty window:



```python
from PyQt6.QtWidgets import QApplication, QWidget

# Only needed for access to command line arguments
import sys

# You need one (and only one) QApplication instanc
# Pass in sys.argv to allow command line arguments
# If you know you won't use command line arguments
app = QApplication(sys.argv)

# Create a Qt widget, which will be our window.
window = QWidget()
window.show()  # IMPORTANT!!!!! Windows are hidden

# Start the event loop.
app.exec()
```

What you'll see will depend on what platform you're running this example on. The image below shows the window as displayed on Windows, macOS and Linux (Ubuntu).



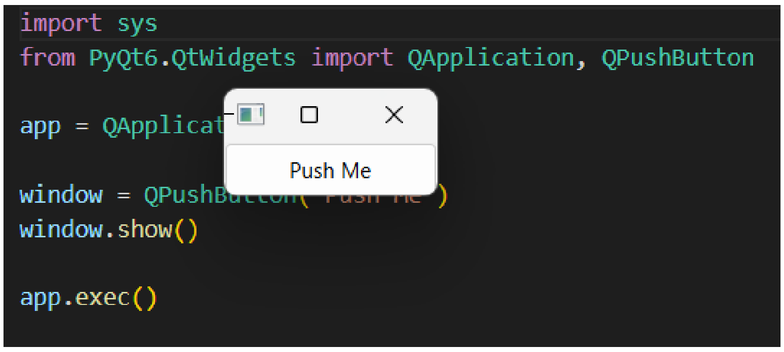
Qt automatically creates a window with the normal window decorations and you can drag it around and resize it like any window. If you want to understand the code line by line then visit this URL.

2.1 QMainWindow

As we discovered in the last part, in Qt any widgets can be windows. For example, if you replace `QtWidget` with `QPushButton`. In the example below, you would get a window with a single push-able button in it.

```python
import sys
from PyQt6.QtWidgets import QApplication, QPushButton

app = QApplication(sys.argv)

window = QPushButton("Push Me")
window.show()

app.exec()
```

The code yields the following output:



```python
import sys
from PyQt6.QtWidgets import QApplication, QPushButton

app = QApplicat

window = QPushButton("Push Me")
window.show()

app.exec()
```

This is neat, but not really very useful; it's rare that you need a UI that consists of only a single control! But, as we'll discover later, the ability to nest widgets within other widgets using layouts means you can construct complex UIs inside an empty `QWidget`.

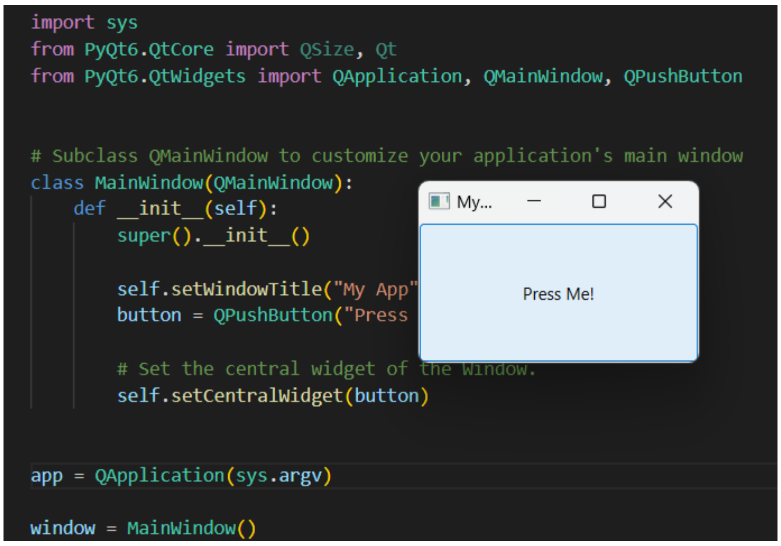
But, Qt already has a solution for you – the `QMainWindow`. This is a pre-made widget which provides a lot of standard window features you'll make use of in your apps, including toolbars, menus, a statusbar, dockable widgets and more.

So lets update the code:

```python
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

# Subclass QMainWindow to customize your application's main window
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")
        button = QPushButton("Press Me!")

        # Set the central widget of the Window.
        self.setCentralWidget(button)

app = QApplication(sys.argv)

window = MainWindow()
window.show()

```

```python
app.exec()
```

The output:



```python
import sys
from PyQt6.QtCore import QSize, Qt
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton


# Subclass QMainWindow to customize your application's main window
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App"
        button = QPushButton("Press

        # Set the central widget of the Window.
        self.setCentralWidget(button)


app = QApplication(sys.argv)

window = MainWindow()
```

For this demo we're using a `QPushButton`. The core Qt widgets are always imported from the `QtWidgets` namespace, as are the `QMainWindow` and `QApplication` classes. When using `QMainWindow` we use `.setCentralWidget` to place a widget (here a `QPushButton`) in the `QMainWindow`. By default it takes the whole of the window. You can look at how to add multiple widgets to windows from this URL.

Note: When you subclass a Qt class you must always call the super `__init__` function to allow Qt to set up the object.

In our `__init__` block we first use `.setWindowTitle()` to change the title of our main window. Then we add our first widget, a `QPushButton`, to the middle of the window. When creating the button you can pass in the text that you want the button to display.

Finally, we call `.setCentralWidget()` on the window. This is a `QMainWindow` specific function that allows you to set the widget that goes in the middle of the window.

Running this code, you will see your window again, but this time with the `QPushButton` widget in the middle. However, pressing the button will do nothing yet.

2.2 Sizing Windows and Widgets

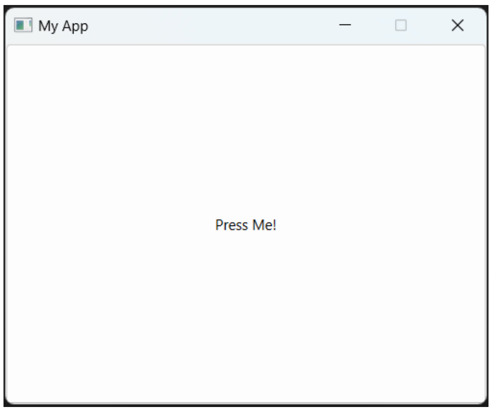
The window is currently freely resizable; if you grab any corner with your mouse, you can drag and resize it to any size you want. While it's good to let your users resize your applications, sometimes you may want to place restrictions on minimum or maximum sizes, or lock a window to a fixed size.

In Qt sizes are defined using a `QSize` object. This accepts width and height parameters in that order. For example, the following will create a fixed size window of 400x300 pixels.

```python
import sys
from PyQt6.QtCore import QSize, Qt
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

# Subclass QMainWindow to customize your application's main window
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")

        self.setFixedSize(QSize(400, 300))

        # Set the central widget of the Window.
        self.setCentralWidget(button)

        # Disable resizing
        self.setWindowFlags(self.windowFlags() & ~Qt.WindowType.
    WindowMaximizeButtonHint)

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

You should get a fixed size window when you set the fixed size using `self.setFixedSize(QSize(400, 300))`. However, there might be some platform-specific behavior or other factors affecting the window's appearance.

To ensure the window is not resizable, you can add the `Qt.WindowFlags` option `Qt.WindowType.WindowMinimizeButtonHint`, which removes the minimize button and disables resizing:



As well as `.setFixedSize()` you can also call `.setMinimumSize()` and `.setMaximumSize()` to set the minimum and maximum sizes respectively.

Note: You can use these size methods on any widget.

3 PyQt6 Signals

So far we've created a window and added a simple push button widget to it, but the button doesn't do anything. What we need is a way to connect the action of pressing the button to making something happen. In Qt, this is provided by signals and slots or events.

3.1 Signals & Slots

Signals are notifications emitted by widgets when something happens. That something can be any number of things, from pressing a button, to the text of an input box changing, to the text of the window changing. Many signals are initiated by user action, but this is not a rule.

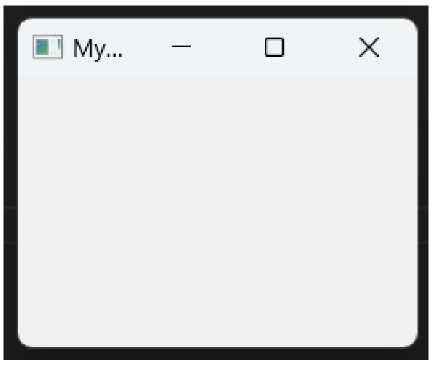
In addition to notifying about something happening, signals can also send data to provide additional context about what happened.

Slots is the name Qt uses for the receivers of signals. In Python any function (or method) in your application can be used as a slot – simply by connecting the signal to it. If the signal sends data, then the receiving function will receive that data too. Many Qt widgets also have their own built-in slots, meaning you can hook Qt widgets together directly.

For this guide we will be using the following code:

```python
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):

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

        self.setWindowTitle("My App")

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Running this code should yield the screen shown in the following figure:

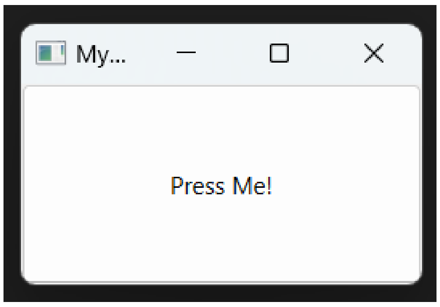


3.2 QPushButton Signals

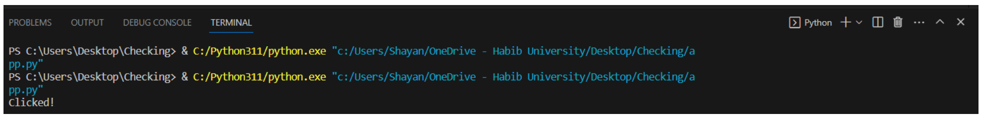
Our simple application currently has a `QMainWindow` with a `QPushButton` set as the central widget. Let's start by hooking up this button to a custom Python method. Here we create a simple custom slot named `the_button_was_clicked`, which accepts the clicked signal from the `QPushButton`.

```python
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)
        button.clicked.connect(self.the_button_was_clicked)

        # Set the central widget of the Window.
        self.setCentralWidget(button)

    def the_button_was_clicked(self):
        print("Clicked!")

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

This should result in the following output screen:



As you can see, the clicking the button will result in a `Clicked!` message in your terminal:



```
PS C:\Users\Desktop\Checking> & C:/Python311/python.exe "c:/Users/Shayan/OneDrive - Habib University/Desktop/Checking/a
pp.py"
PS C:\Users\Desktop\Checking> & C:/Python311/python.exe "c:/Users/Shayan/OneDrive - Habib University/Desktop/Checking/a
pp.py"
Clicked!
```

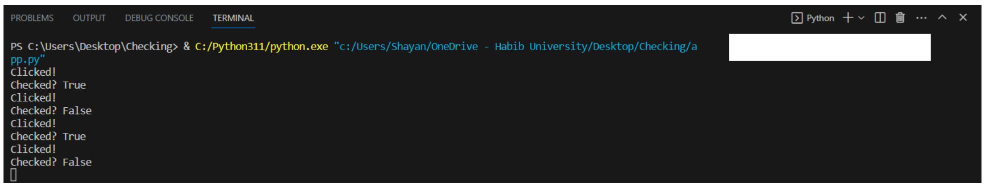
3.2.1 Receiving Data

We've heard already that signals can also send data to provide more information about what has just happened. The `.clicked` signal is no exception, also providing a checked (or toggled) state for the button. For normal buttons this is always `False`, so our first slot ignored this data. However, we can make our button checkable and see the effect.

In the following example, we add a second slot which outputs the checkstate:

```python
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)
        button.clicked.connect(self.the_button_was_clicked)
        button.clicked.connect(self.the_button_was_toggled)

        self.setCentralWidget(button)

    def the_button_was_clicked(self):
        print("Clicked!")

    def the_button_was_toggled(self, checked):
        print("Checked?", checked)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Run the code. Now, if you press the button you'll see it highlighted as checked. Press it again to release it. Look for the check state in the console.



```
PS C:\Users\Desktop\Checking> & C:/Python311/python.exe "c:/Users/Shayan/OneDrive - Habib University/Desktop/Checking/a
pp.py"
Clicked!
Checked? True
Clicked!
Checked? False
Clicked!
Checked? True
Clicked!
Checked? False
```

3.2.2 Storing data

Often it is useful to store the current state of a widget in a Python variable. This allows you to work with the values like any other Python variable and without accessing the original widget. You can either store these values as individual variables or use a dictionary if you prefer. In the next example, we store the checked value of our button in a variable called `button_is_checked` on `self`.

```
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.button_is_checked = True

        self.setWindowTitle("My App")

        button = QPushButton("Press Me!")
        button.setCheckable(True)
        button.clicked.connect(self.the_button_was_toggled)
        button.setChecked(self.button_is_checked)

        self.setCentralWidget(button)

    def the_button_was_toggled(self, checked):
        self.button_is_checked = checked

        print(self.button_is_checked)


app = QApplication(sys.argv)

```

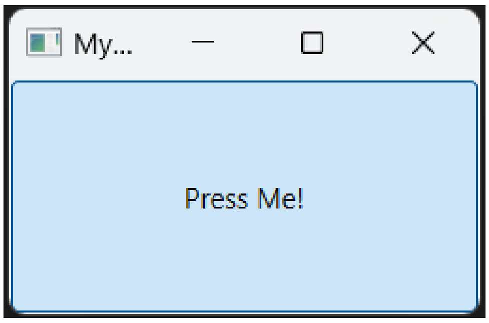


Figure 1: State is `True`

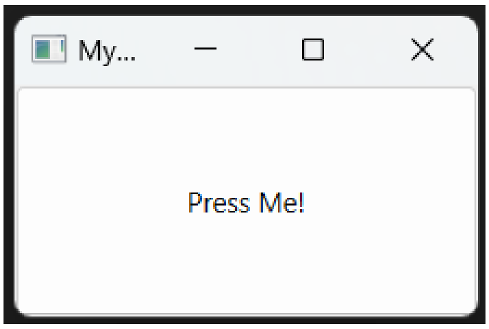


Figure 2: State is `False`

```python
window = MainWindow()
window.show()

app.exec()
```

First, we set the default value for our variable (to True), then use the default value to set the initial state of the widget. When the widget state changes, we receive the signal and update the variable to match. The following image shows the states and the resulting screen of the GUI:

You can use this same pattern with any PyQt widgets. If a widget does not provide a signal that sends the current state, you will need to retrieve the value from the widget directly in your handler. For example, here we're checking the checked state in a pressed handler.

```python
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.button_is_checked = True

        self.setWindowTitle("My App")

        self.button = QPushButton("Press Me!")
        self.button.setCheckable(True)
        self.button.released.connect(self.the_button_was_released)
        self.button.setChecked(self.button_is_checked)

        self.setCentralWidget(self.button)

    def the_button_was_released(self):
        self.button_is_checked = self.button.isChecked()

        print(self.button_is_checked)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Note: We need to keep a reference to the button on self so we can access it in our

slot.

The released signal fires when the button is released, but does not send the check state, so instead we use `.isChecked()` to get the check state from the button in our handler.

3.2.3 Changing the interface

So far we've seen how to accept signals and print output to the console. But how about making something happen in the interface when we click the button? Let's update our slot method to modify the button, changing the text and disabling the button so it is no longer clickable. We'll also turn off the checkable state for now.

```python
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        self.button = QPushButton("Press Me!")
        self.button.clicked.connect(self.the_button_was_clicked)

        self.setCentralWidget(self.button)

    def the_button_was_clicked(self):
        self.button.setText("You already clicked me.")
        self.button.setEnabled(False)

        # Also change the window title.
        self.setWindowTitle("My Oneshot App")

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Again, because we need to be able to access the `button` in our `the_button_was_clicked` method, we keep a reference to it on `self`. The text of the button is changed by passing a `str` to `.setText()`. To disable a button call `.setEnabled()` with `False`.

Run it! If you click the button the text will change and the button will become unclickable as shown below:

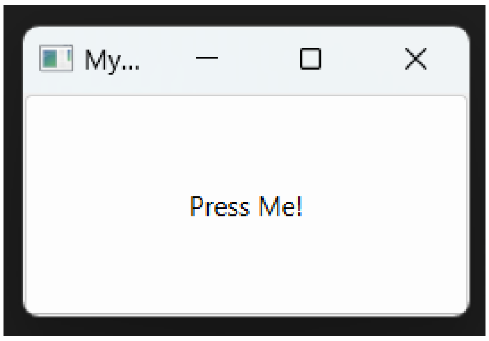


Figure 3: Before Clicking.

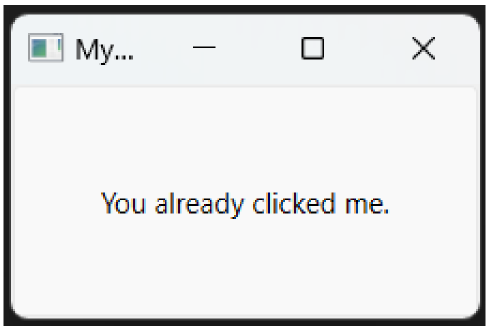


Figure 4: After Clicking.

You're not restricted to changing the button that triggers the signal, you can do anything you want in your slot methods. For example, try adding the following line to `the_button_was_clicked` method to also change the window title.

```python
self.setWindowTitle("A new window title")
```

Most widgets have their own signals – and the `QMainWindow` we're using for our window is no exception. In the following more complex example, we connect the `.windowTitleChanged` signal on the `QMainWindow` to a custom slot method.

3.2.4 Connecting widgets together directly

So far we've seen examples of connecting widget signals to Python methods. When a signal is fired from the widget, our Python method is called and receives the data from the signal. But you don't always need to use a Python function to handle signals – you can also connect Qt widgets directly to one another.

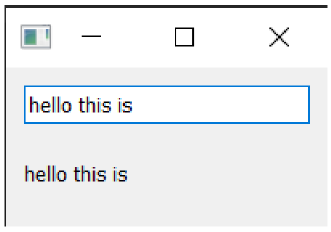
In the following example, we add a `QLineEdit` widget and a `QLabel` to the window. In the `__init__` for the window we connect our line edit `.textChanged` signal to the `.setText` method on the `QLabel`. Now any time the text changes in the `QLineEdit` the `QLabel` will receive that text to it's `.setText` method.

```python
from PyQt6.QtWidgets import QApplication, QMainWindow, QLabel, QLineEdit,
    QVBoxLayout, QWidget

import sys


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        self.label = QLabel()

        self.input = QLineEdit()
        self.input.textChanged.connect(self.label.setText)

        layout = QVBoxLayout()
        layout.addWidget(self.input)
        layout.addWidget(self.label)

        container = QWidget()
        container.setLayout(layout)

        # Set the central widget of the Window.
        self.setCentralWidget(container)


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Notice that in order to connect the input to the label, the input and label must both be defined. This code adds the two widgets to a layout, and sets that on the

window. We'll cover layouts in detail later, you can ignore it for now.

Run it! Type some text in the upper box, and you'll see it appear immediately on the label as shown below:


hello this is

hello this is

3.2.5 Events

Every interaction the user has with a Qt application is an event. There are many types of event, each representing a different type of interaction. Qt represents these events using event objects which package up information about what happened. These events are passed to specific event handlers on the widget where the interaction occurred.

Event handlers are defined just like any other method, but the name is specific for the type of event they handle.

One of the main events which widgets receive is the `QMouseEvent`. `QMouseEvent` events are created for each and every mouse movement and button click on a widget. The following event handlers are available for handling mouse events –

Event handler	Event type moved
mouseMoveEvent	Mouse moved
mousePressEvent	Mouse button pressed
mouseReleaseEvent	Mouse button released
mouseDoubleClickEvent	Double click detected

For example, clicking on a widget will cause a `QMouseEvent` to be sent to the `.mousePressEvent` event handler on that widget. This handler can use the event object to find out information about what happened, such as what triggered the event and where specifically it occurred.

You can intercept events by sub-classing and overriding the handler method on the class. You can choose to filter, modify, or ignore events, passing them up to the normal handler for the event by calling the parent class function with `super()`. These could be added to your main window class as follows. In each case `e` will receive the incoming event.

```python
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QLabel, QMainWindow, QTextEdit


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.label = QLabel("Click in this window")
        self.setCentralWidget(self.label)

    def mouseMoveEvent(self, e):
        self.label.setText("mouseMoveEvent")

    def mousePressEvent(self, e):
        self.label.setText("mousePressEvent")

    def mouseReleaseEvent(self, e):
        self.label.setText("mouseReleaseEvent")

    def mouseDoubleClickEvent(self, e):
        self.label.setText("mouseDoubleClickEvent")


app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Run it! Try moving and clicking (and double-clicking) in the window and watch the events appear.

You'll notice that mouse move events are only registered when you have the button pressed down. You can change this by calling `self.setMouseTracking(True)` on the window. You may also notice that the press (click) and double-click events both fire when the button is pressed down. Only the release event fires when the button is released. Typically to register a click from a user you should watch for both the mouse down and the release.

3.2.6 Mouse events

All mouse events in Qt are tracked with the `QMouseEvent` object, with information about the event being readable from the following event methods:

Method	Returns
`.button()`	Specific button that triggered this event
`.buttons()`	State of all mouse buttons (OR'ed flags)
`.position()`	Widget-relative position as a `QPoint` *integer*

You can use these methods within an event handler to respond to different events differently, or ignore them completely.

For example, the following allows us to respond differently to a left, right or middle click on the window.

```python
from PyQt6.QtWidgets import QApplication, QMainWindow, QLabel, QLineEdit,
    QVBoxLayout, QWidget
from PyQt6.QtCore import Qt
import sys

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        self.label = QLabel()

        self.input = QLineEdit()
        self.input.textChanged.connect(self.label.setText)

        layout = QVBoxLayout()
        layout.addWidget(self.input)
        layout.addWidget(self.label)

        container = QWidget()
        container.setLayout(layout)

        # Set the central widget of the Window.
        self.setCentralWidget(container)
    def mousePressEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            # handle the left-button press in here
            self.label.setText("mousePressEvent LEFT")

        elif e.button() == Qt.MouseButton.MiddleButton:
            # handle the middle-button press in here.
            self.label.setText("mousePressEvent MIDDLE")

        elif e.button() == Qt.MouseButton.RightButton:
            # handle the right-button press in here.
            self.label.setText("mousePressEvent RIGHT")

    def mouseReleaseEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            self.label.setText("mouseReleaseEvent LEFT")

        elif e.button() == Qt.MouseButton.MiddleButton:
            self.label.setText("mouseReleaseEvent MIDDLE")

        elif e.button() == Qt.MouseButton.RightButton:
            self.label.setText("mouseReleaseEvent RIGHT")

    def mouseDoubleClickEvent(self, e):
        if e.button() == Qt.MouseButton.LeftButton:
            self.label.setText("mouseDoubleClickEvent LEFT")

        elif e.button() == Qt.MouseButton.MiddleButton:
            self.label.setText("mouseDoubleClickEvent MIDDLE")

        elif e.button() == Qt.MouseButton.RightButton:
            self.label.setText("mouseDoubleClickEvent RIGHT")

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

The button identifiers are defined in the Qt namespace, as follows:

Identifier	Value (binary)	Represents
Qt.MouseButton.NoButton	0 (000)	No button pressed, or the event is not related to button press.
Qt.MouseButton.LeftButton	1 (001)	The left button is pressed
Qt.MouseButton.RightButton	2 (010)	The right button is pressed.
Qt.MouseButton.MiddleButton	4 (100)	The middle button is pressed.

4 PyQt6 Widgets

In Qt (and most User Interfaces) 'widget' is the name given to a component of the UI that the user can interact with. User interfaces are made up of multiple widgets, arranged within the window.

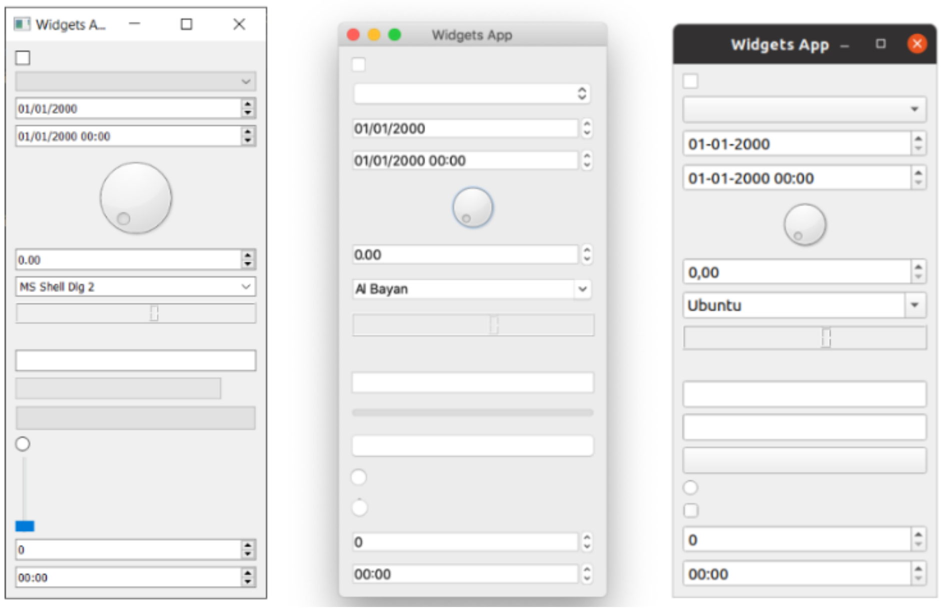
Qt comes with a large selection of widgets available, and even allows you to create your own custom and customized widgets.

4.1 A quick demo

First, let's have a look at some of the most common PyQt widgets. The following code creates a range of PyQt widgets and adds them to a window layout so you can see them together.

```python
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import (
    QApplication,
    QCheckBox,
    QComboBox,
    QDateEdit,
    QDateTimeEdit,
    QDial,
    QDoubleSpinBox,
    QFontComboBox,
    QLabel,
    QLCDNumber,
    QLineEdit,
    QMainWindow,
    QProgressBar,
    QPushButton,
    QRadioButton,
    QSlider,
    QSpinBox,
    QTimeEdit,
    QVBoxLayout,
    QWidget,
)


# Subclass QMainWindow to customize your application's main window
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Widgets App")

        layout = QVBoxLayout()
        widgets = [
            QCheckBox,
            QComboBox,
            QDateEdit,
            QDateTimeEdit,
            QDial,
            QDoubleSpinBox,
            QFontComboBox,
            QLCDNumber,
            QLabel,
            QLineEdit,
            QProgressBar,
            QPushButton,
            QRadioButton,
            QSlider,
            QSpinBox,
            QTimeEdit,
        ]

        for w in widgets:
            layout.addWidget(w())

        widget = QWidget()
        widget.setLayout(layout)

        # Set the central widget of the Window. Widget will expand
        # to take up all the space in the window by default.
        self.setCentralWidget(widget)


app = QApplication(sys.argv)
window = MainWindow()
window.show()

app.exec()
```

Run it! You'll see a window appear containing all the widgets we've created. You can see this in the screenshot below:



Let's have a look at all the example widgets, from top to bottom, as shown in below:

Widget	What it does
QCheckbox	A checkbox
QComboBox	A dropdown list box
QDateEdit	For editing dates and datetimes
QDateTimeEdit	For editing dates and datetimes
QDial	Rotatable dial
QDoubleSpinbox	A number spinner for floats
QFontComboBox	A list of fonts
QLCDNumber	A quite ugly LCD display
QLabel	Just a label, not interactive
QLineEdit	Enter a line of text
QProgressBar	A progress bar
QPushButton	A button
QRadioButton	A toggle set, with only one active item
QSlider	A slider
QSpinBox	An integer spinner
QTimeEdit	For editing times

There are far more widgets than this, but they don't fit so well! You can see them all by checking the PyQt6 widgets documentation.

Next, we'll step through some of the most commonly used widgets and look at them in more detail. To experiment with the widgets we'll need a simple application to put them in. Save the following code to a file named `app.py` and run it to make sure it's working.

```python
import sys
from PyQt6.QtWidgets import (
    QMainWindow, QApplication,
    QLabel, QCheckBox, QComboBox, QListWidget, QLineEdit,
    QLineEdit, QSpinBox, QDoubleSpinBox, QSlider
)
from PyQt6.QtCore import Qt

class MainWindow(QMainWindow):

    def __init__(self):
        super(MainWindow, self).__init__()

        self.setWindowTitle("My App")


app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

In the code above we've imported a number of Qt widgets. Now we'll step through each of those widgets in turn, adding them to our application and seeing how they behave.

4.2 QLabel

We'll start the tour with QLabel, arguably one of the simplest widgets available in the Qt toolbox. This is a simple one-line piece of text that you can position in your application. You can set the text by passing in a str as you create it:

```python
widget = QLabel("Hello")
```

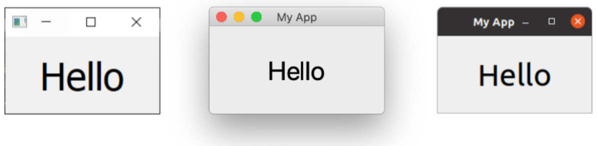
Or, by using the `.setText()` method:

```python
widget = QLabel("1")  # The label is created with the text 1.
widget.setText("2")   # The label now shows 2.
```

You can also adjust font parameters, such as the size of the font or the alignment of text in the widget.

```python
import sys
from PyQt6.QtWidgets import (
    QMainWindow, QApplication,
    QLabel, QCheckBox, QComboBox, QListWidget, QLineEdit,
    QLineEdit, QSpinBox, QDoubleSpinBox, QSlider
)
from PyQt6.QtCore import Qt
class MainWindow(QMainWindow):

    def __init__(self):
        super(MainWindow, self).__init__()

        self.setWindowTitle("My App")

        widget = QLabel("Hello")
        font = widget.font()
        font.setPointSize(30)
        widget.setFont(font)
        widget.setAlignment(Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.
    AlignVCenter)

        self.setCentralWidget(widget)

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

Here is what the output of the code looks like:



The alignment is specified by using a flag from the `Qt.` namespace. The flags available for horizontal alignment are:

PyQt6 flag (long name)	Behavior
`Qt.AlignmentFlag.AlignLeft`	Aligns with the left edge.
`Qt.AlignmentFlag.AlignRight`	Aligns with the right edge.
`Qt.AlignmentFlag.AlignHCenter`	Centers horizontally in the available space.
`Qt.AlignmentFlag.AlignJustify`	Justifies the text in the available space.

The flags available for vertical alignment are:

PyQt6 flag (long name)	Behavior
Qt.AlignmentFlag.AlignTop	Aligns with the top.
Qt.AlignmentFlag.AlignBottom	Aligns with the bottom.
Qt.AlignmentFlag.AlignVCenter	Centers vertically in the available space.

You can combine flags together using pipes (|), however note that you can only use vertical or horizontal alignment flag at a time.

```
align_top_left = Qt.AlignmentFlag.AlignLeft | Qt.AlignmentFlag.AlignTop
```

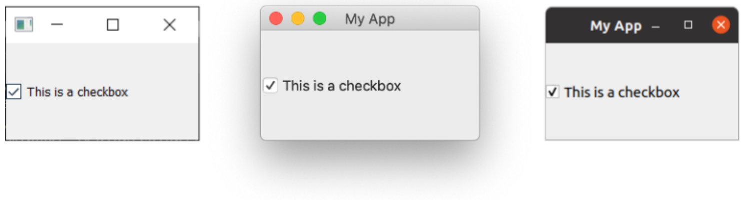
You can learn more about alignment in the PyQt6 documentation here.

4.3 QCheckBox

The next widget to look at is `QCheckBox()` which, as the name suggests, presents a checkable box to the user. However, as with all Qt widgets there are number of configurable options to change the widget behaviors.

```python
import sys
from PyQt6.QtWidgets import (
    QMainWindow, QApplication,
    QLabel, QCheckBox, QComboBox, QListWidget, QLineEdit,
    QLineEdit, QSpinBox, QDoubleSpinBox, QSlider
)
from PyQt6.QtCore import Qt

class MainWindow(QMainWindow):

    def __init__(self):
        super(MainWindow, self).__init__()

        self.setWindowTitle("My App")

        widget = QCheckBox()
        widget.setCheckState(Qt.CheckState.Checked)

        # For tristate: widget.setCheckState(Qt.PartiallyChecked)
        # Or: widget.setTriState(True)
        widget.stateChanged.connect(self.show_state)

        self.setCentralWidget(widget)


    def show_state(self, s):
        print(s == Qt.CheckState.Checked)
        print(s)

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

If you run the code you will see the following output:



You can set a checkbox state programmatically using `.setChecked` or `.setCheckState`. The former accepts either `True` or `False` representing checked or unchecked respectively. However, with `.setCheckState` you also specify a particular checked state using a `Qt.` namespace flag:

PyQt6 flag (long name)	Behavior
`Qt.CheckState.Unchecked`	Item is unchecked
`Qt.CheckState.PartiallyChecked`	Item is partially checked
`Qt.CheckState.Checked`	Item is checked

A checkbox that supports a partially-checked (`Qt.CheckState.PartiallyChecked`) state is commonly referred to as 'tri-state', that is being neither on nor off. A checkbox in this state is commonly shown as a greyed out checkbox, and is commonly used in hierarchical checkbox arrangements where sub-items are linked to parent checkboxes.

You can learn more about the functionality of checkboxes by checking out the PyQt6 documentation here.

4.4 QComboBox

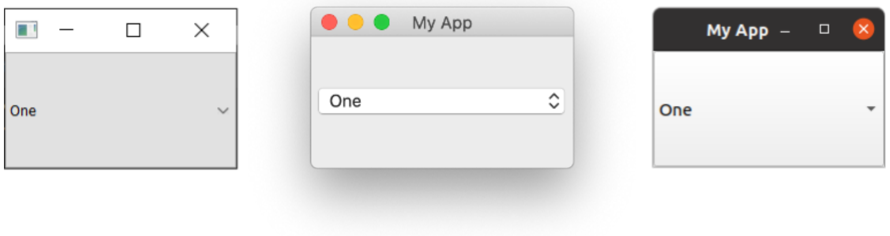
The `QComboBox` is a drop down list, closed by default with an arrow to open it. You can select a single item from the list, with the currently selected item being shown as a label on the widget. The combo box is suited to selection of a choice from a long list of options.

You have probably seen the combo box used for selection of font faces, or size, in word processing applications. Although Qt actually provides a specific font-selection combo box as `QFontComboBox`.

You can add items to a `QComboBox` by passing a list of strings to `.addItems()`. Items will be added in the order they are provided.

```python
import sys
from PyQt6.QtWidgets import (
    QMainWindow, QApplication,
    QLabel, QCheckBox, QComboBox, QListWidget, QLineEdit,
    QLineEdit, QSpinBox, QDoubleSpinBox, QSlider
)
from PyQt6.QtCore import Qt

class MainWindow(QMainWindow):

    def __init__(self):
        super(MainWindow, self).__init__()

        self.setWindowTitle("My App")

        widget = QComboBox()
        widget.addItems(["One", "Two", "Three"])

        # Sends the current index (position) of the selected item.
        widget.currentIndexChanged.connect( self.index_changed )

        # There is an alternate signal to send the text.
        widget.currentTextChanged.connect( self.text_changed )

        self.setCentralWidget(widget)


    def index_changed(self, i): # i is an int
        print(i)

    def text_changed(self, s): # s is a str
        print(s)

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

Here is what the output looks like:



The .currentIndexChanged signal is triggered when the currently selected item is updated, by default passing the index of the selected item in the list. There is also a .currentTextChanged signal which instead provides the label of the currently selected item, which is often more useful.

QComboBox can also be editable, allowing users to enter values not currently in the list and either have them inserted, or simply used as a value. To make the box editable:

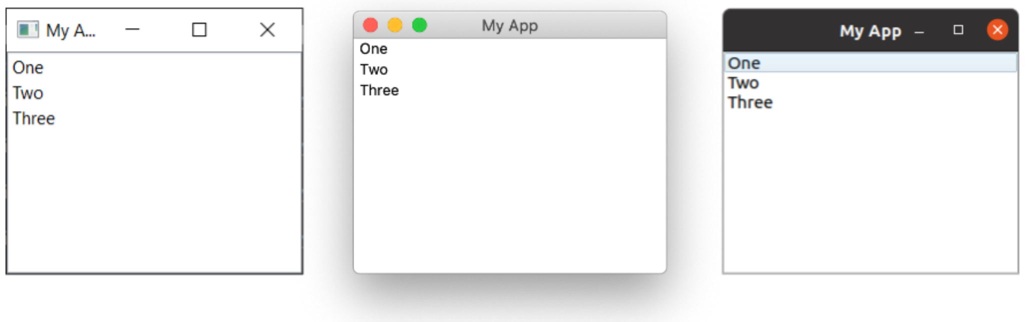
```python
widget.setEditable(True)
```

4.5 QListWidget

This widget is similar to QComboBox, except options are presented as a scrollable list of items. It also supports selection of multiple items at once. A QListWidget offers an currentItemChanged signal which sends the QListWidgetItem (the element of the list widget), and a currentTextChanged signal which sends the text of the current item.

```python
import sys
from PyQt6.QtWidgets import (
    QMainWindow, QApplication,
    QLabel, QCheckBox, QComboBox, QListWidget, QLineEdit,
    QLineEdit, QSpinBox, QDoubleSpinBox, QSlider
)
from PyQt6.QtCore import Qt

class MainWindow(QMainWindow):

    def __init__(self):
        super(MainWindow, self).__init__()

        self.setWindowTitle("My App")

        widget = QListWidget()
        widget.addItems(["One", "Two", "Three"])

        widget.currentItemChanged.connect(self.index_changed)
        widget.currentTextChanged.connect(self.text_changed)

        self.setCentralWidget(widget)


    def index_changed(self, i): # Not an index, i is a QListWidgetItem
        print(i.text())

    def text_changed(self, s): # s is a str
        print(s)

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

Here is what the output looks like:

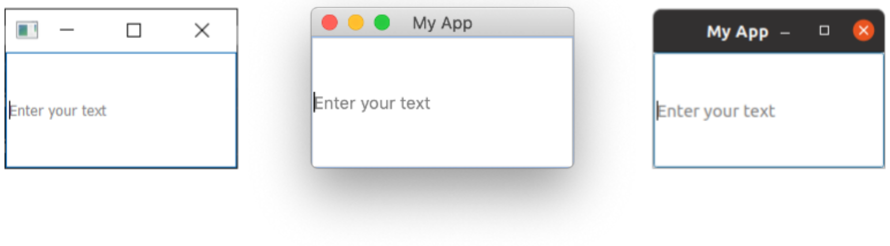


4.6 QLineEdit

The `QLineEdit` widget is a simple single-line text editing box, into which users can type input. These are used for form fields, or settings where there is no restricted list of valid inputs. For example, when entering an email address, or computer name.

```python
import sys
from PyQt6.QtWidgets import (
    QMainWindow, QApplication,
    QLabel, QCheckBox, QComboBox, QListWidget, QLineEdit,
    QLineEdit, QSpinBox, QDoubleSpinBox, QSlider
)
from PyQt6.QtCore import Qt

class MainWindow(QMainWindow):

    def __init__(self):
        super(MainWindow, self).__init__()

        self.setWindowTitle("My App")

        widget = QLineEdit()
        widget.setMaxLength(10)
        widget.setPlaceholderText("Enter your text")

        #widget.setReadOnly(True) # uncomment this to make readonly

        widget.returnPressed.connect(self.return_pressed)
        widget.selectionChanged.connect(self.selection_changed)
        widget.textChanged.connect(self.text_changed)
        widget.textEdited.connect(self.text_edited)

        self.setCentralWidget(widget)


    def return_pressed(self):
        print("Return pressed!")
        self.centralWidget().setText("BOOM!")

    def selection_changed(self):
        print("Selection changed")
        print(self.centralWidget().selectedText())

    def text_changed(self, s):
        print("Text changed...")
        print(s)

    def text_edited(self, s):
        print("Text edited...")
        print(s)

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

Here is what the output looks like:



As demonstrated in the above code, you can set a maximum length for the text in a line edit.

You can learn more about Widgets in the PyQt6 documentation here.

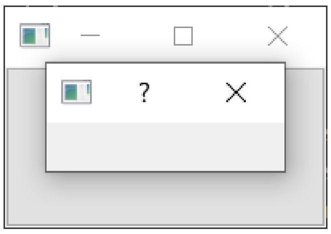
5 PyQt6 Dialogs

Dialogs are useful GUI components that allow you to communicate with the user. They are small blocking windows that sit in front of the main application until they are dismissed. Qt provides a number of 'special' built-in dialogs for the most common use-cases, allowing you to provide a platform-native user experience.

Let's create our own `QDialog`:

```python
import sys

from PyQt6.QtWidgets import QApplication, QDialog, QMainWindow, QPushButton

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press me for a dialog!")
        button.clicked.connect(self.button_clicked)
        self.setCentralWidget(button)

    def button_clicked(self, s):
        print("click", s)

        dlg = QDialog(self)
        dlg.setWindowTitle("HELLO!")
        dlg.exec()

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Output:



In the `button_clicked` method (which receives the signal from the button press) we create the dialog instance, passing our `QMainWindow` instance as a parent. This will make the dialog a modal window of `QMainWindow`. This means the dialog will completely block interaction with the parent window.

Once we have created the dialog, we start it using `.exec()` - just like we did for `QApplication` to create the main event loop of our application. That's not a coincidence: when you exec the `QDialog` an entirely new event loop - specific for the dialog - is created.

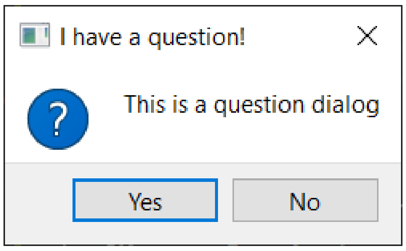
Note: The `QDialog` completely blocks your application execution. Don't start a dialog and expect anything else to happen anywhere else in your app. We'll see later how you can use threads & processes to get you out of this pickle.

The first step in creating a dialog button box is to define the buttons want to show, using namespace attributes from `QDialogButtonBox`. The full list of buttons available is below. You can also construct a line of multiple buttons by OR-ing them together using a pipe (|). Qt will handle the order automatically, according to platform standards. You can also tweak the icon shown on the dialog by setting the icon with one of the following. See details for all these features using this URL.

Now let's look at an example that uses these concepts:

```python
import sys

from PyQt6.QtWidgets import QApplication, QDialog, QMainWindow, QMessageBox,
    QPushButton


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("My App")

        button = QPushButton("Press me for a dialog!")
        button.clicked.connect(self.button_clicked)
        self.setCentralWidget(button)

    def button_clicked(self, s):
        dlg = QMessageBox(self)
        dlg.setWindowTitle("I have a question!")
        dlg.setText("This is a question dialog")
        dlg.setStandardButtons(QMessageBox.StandardButton.Yes | QMessageBox.
    StandardButton.No)
        dlg.setIcon(QMessageBox.Icon.Question)
        button = dlg.exec()

        if button == QMessageBox.Yes:
            print("Yes!")
        else:
            print("No!")

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Output:


I have a question!

This is a question dialog

6 Multi-Window PyQt6

Other than dialog windows, quite often you will want to open a second window in an application, without interrupting the main window.

6.1 Creating a Window

In Qt any widget without a parent is a window. This means, to show a new window you just need to create a new instance of a widget. This can be any widget type (technically any subclass of `QWidget`) including another `QMainWindow` if you prefer.

Note: There is no restriction on the number of `QMainWindow` instances you can have. If you need toolbars or menus on your second window you will have to use a `QMainWindow` to achieve this. This can get confusing for users however, so make sure it's necessary.

```python
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton, QLabel,
    QVBoxLayout, QWidget
import sys

class AnotherWindow(QWidget):
    """
    This "window" is a QWidget. If it has no parent, it
    will appear as a free-floating window as we want.
    """
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window")
        layout.addWidget(self.label)
        self.setLayout(layout)

class MainWindow(QMainWindow):

    def __init__(self):
        super().__init__()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.show_new_window)
        self.setCentralWidget(self.button)

    def show_new_window(self, checked):
        self.w = AnotherWindow()
        self.w.show()

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

If you run this, you'll see the main window. Clicking the button may show the second window. However, what happens if you click the button again? The window will be re-created! This new window will replace the old in the `self.w` variable, and – because there is now no reference to it – the previous window will be destroyed.

6.2 Persistent Windows

So far we've looked at how to create new windows on demand. However, sometimes you have a number of standard application windows. In this case rather than create the windows when you want to show them, it can often make more sense to create them at start-up, then use `.show()` to display them when needed.

In the following example we create our external window in the `__init__` block for the main window, and then our `show_new_window` method simply calls `self.w.show()` to display it.

```python
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton, QLabel,
    QVBoxLayout, QWidget
import sys

from random import randint

class AnotherWindow(QWidget):
    """
    This "window" is a QWidget. If it has no parent, it
    will appear as a free-floating window as we want.
    """
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window % d" % randint(0,100))
        layout.addWidget(self.label)
        self.setLayout(layout)

class MainWindow(QMainWindow):

    def __init__(self):
        super().__init__()
        self.w = AnotherWindow()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.show_new_window)
        self.setCentralWidget(self.button)

    def show_new_window(self, checked):
        self.w.show()

app = QApplication(sys.argv)
w = MainWindow()
w.show()
app.exec()
```

If you run this, clicking on the button will show the window as before. However, note that the window is only created once and calling `.show()` on an already visible window has no effect.

7 Linking `.ui` files

Once you've created your own `.ui` file using the Qt

```python
import sys
from PyQt6 import QtWidgets
from PyQt6 import uic


class UI(QtWidgets.QMainWindow):
    def __init__(self):
        # Call the inherited classes __init__ method
        super(UI, self).__init__()
        # Load the .ui file
        uic.loadUi('your_file.ui', self)
        # Show the GUI
        self.show()


# Create an instance of QtWidgets.QApplication
app = QtWidgets.QApplication(sys.argv)
window = UI()  # Create an instance of our class
app.exec()  # Start the application
```

2

[2]Created by CS Research Assistants: Shafaq Fatima Mughal & Shayan Aamir