Building Engineering Applications with Python and PyQt6

Contents

[Introduction to Python and PyQt6 2](#_Toc157791206)

[Introduction 2](#_Toc157791207)

[Why Python and PyQt6? 2](#_Toc157791208)

[What This Book Covers 2](#_Toc157791209)

[How to Use This Book 3](#_Toc157791210)

[Let's Get Started! 3](#_Toc157791211)

[Python basics and syntax 4](#_Toc157791212)

[What is PyQt6? 4](#_Toc157791213)

[Installing PyQt6 4](#_Toc157791214)

[Setting up your development environment 4](#_Toc157791215)

[Getting Started with PyQt6 4](#_Toc157791216)

[Creating your first PyQt6 application 4](#_Toc157791217)

[Understanding PyQt6 widgets 4](#_Toc157791218)

[Layout management in PyQt6 4](#_Toc157791219)

[Event handling in PyQt6 4](#_Toc157791220)

[PyQt6 Widgets in Depth 4](#_Toc157791221)

[Commonly used widgets (buttons, labels, text boxes, etc.) 4](#_Toc157791222)

[Advanced widgets (tables, trees, dialogs, etc.) 4](#_Toc157791223)

[Custom widgets and subclassing 4](#_Toc157791224)

[Styling and Theming in PyQt6 4](#_Toc157791225)

[Using style sheets to customize widget appearance 4](#_Toc157791226)

[Applying themes to your PyQt6 application 4](#_Toc157791227)

[PyQt6 Signals and Slots 4](#_Toc157791228)

[Understanding signals and slots in PyQt6 4](#_Toc157791229)

[Connecting signals to slots 4](#_Toc157791230)

[Emitting custom signals 4](#_Toc157791231)

[Integrating Scientific Libraries with PyQt6 4](#_Toc157791232)

[Introduction to popular scientific libraries (NumPy, SciPy, Matplotlib, etc.) 4](#_Toc157791233)

[Using scientific libraries in PyQt6 applications 4](#_Toc157791234)

[Visualizing scientific data with PyQt6 and Matplotlib 4](#_Toc157791235)

[Signal Processing with PyQt6 4](#_Toc157791236)

[Fundamentals of signal processing 5](#_Toc157791237)

[Introduction to PyQt6's signal processing capabilities 5](#_Toc157791238)

[Processing and filtering signals in real-time applications 5](#_Toc157791239)

[Real-Time Data Visualization with PyQt6 5](#_Toc157791240)

[Introduction to real-time data visualization 5](#_Toc157791241)

[Updating PyQt6 widgets dynamically 5](#_Toc157791242)

[Building real-time applications with PyQt6 5](#_Toc157791243)

[PyQt6 and Data Analysis 5](#_Toc157791244)

[Loading and processing data in PyQt6 applications 5](#_Toc157791245)

[Analysing data using scientific libraries 5](#_Toc157791246)

[Presenting data analysis results with PyQt6 5](#_Toc157791247)

[Advanced PyQt6 Techniques 5](#_Toc157791248)

[Multi-threading in PyQt6 applications 5](#_Toc157791249)

[Internationalization and localization 5](#_Toc157791250)

[Packaging and distributing PyQt6 applications 5](#_Toc157791251)

[Case Studies and Practical Examples 5](#_Toc157791252)

[Building a scientific data analysis tool with PyQt6 5](#_Toc157791253)

[Developing a real-time monitoring application 5](#_Toc157791254)

[Showcasing advanced PyQt6 features in real-world scenarios 5](#_Toc157791255)

# Introduction to Python and PyQt6

# Introduction

Welcome to the world of Python and PyQt6! In this book, we will embark on an exciting journey to explore the powerful combination of Python programming and PyQt6, a set of Python bindings for the Qt application framework. Whether you're a beginner looking to dive into GUI (Graphical User Interface) development or an experienced Python developer seeking to expand your skill set, this book will provide you with the knowledge and tools you need to create dynamic and interactive applications.

The book as a journey focuses on building representative applications around various interesting topic of engineering and disciplines so as to provide a good head start. Engineering applications such as Artificial Intelligence, IOT, signal processing, embedded systems, real time communication are also covered through examples that are clearly starting from scratch taking the reader through the journey to build functional utilities for their respective use cases.

## Why Python and PyQt6?

Python has emerged as one of the most popular programming languages in recent years, known for its simplicity, readability, and versatility. Its extensive standard library and vibrant community make it an ideal choice for a wide range of applications, from web development to data analysis, and of course, GUI programming.

------------------------------------------------------------------Updated

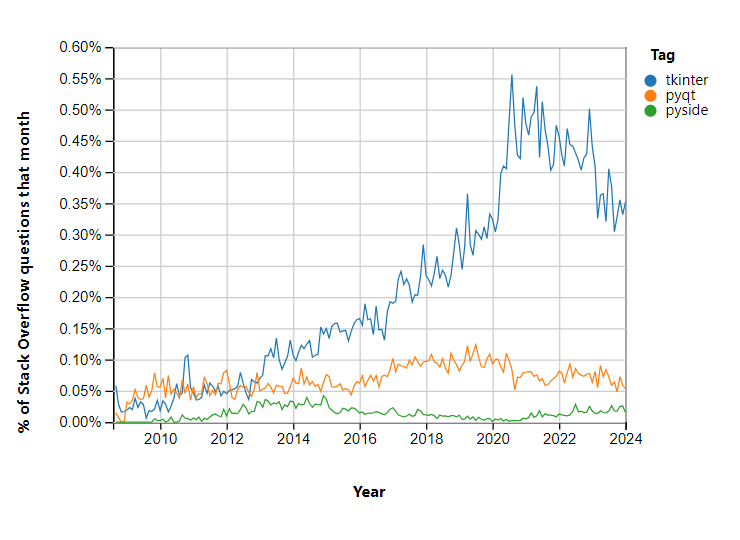
Python becomes one of the most widely used programming languages in recent times. From the stack overflow trends you can see the comparison with other languages.

A graph of different colored lines

Description automatically generated

PyQt6, built on top of the Qt framework, brings the power and flexibility of Qt to Python developers. Qt is a comprehensive cross-platform toolkit used for developing applications with native-looking user interfaces. With PyQt6, developers can leverage the rich features of Qt while enjoying the simplicity and elegance of Python.

There are other GUI libraries available in python like tkinter, pyqt, pyside, etc. PyQt6is one of the famous library used for developing good looking GUI. PyQt6 provides features to create rich looking application with wide range of components. Below is the trends in using GUI library in python. When compared to other libraries PyQt6 is helpful in creating complex applications and gives more provision for customizations. Below is the current trend in market for python GUI libraries tkinter, PyQt and pySide.



## What This Book Covers

This book is designed to be a comprehensive guide to PyQt6, covering everything from the basics of Python and PyQt6 to advanced topics such as real-time data visualization and signal processing. Here's an overview of what you can expect to learn:

* **Python Fundamentals**: We'll start by covering the basics of Python programming, ensuring that you have a solid foundation before diving into PyQt6.
* **Getting Started with PyQt6**: You'll learn how to set up your development environment and create your first PyQt6 application.
* **Understanding PyQt6 Widgets**: We'll explore PyQt6's extensive collection of widgets and learn how to use them to build powerful GUIs.
* **Styling and Theming**: You'll discover how to customize the appearance of your PyQt6 applications using style sheets and themes.
* **Signals and Slots**: We'll delve into PyQt6's signal and slot mechanism, a powerful feature for handling events and communication between objects.
* **Integrating Scientific Libraries**: You'll learn how to integrate popular scientific libraries such as NumPy and Matplotlib with PyQt6 for data analysis and visualization.
* **Signal Processing**: We'll explore how to process and filter signals in real-time applications using PyQt6.
* **Real-Time Data Visualization**: You'll discover techniques for updating PyQt6 widgets dynamically to visualize real-time data.
* **Advanced Techniques**: We'll cover advanced topics such as multi-threading, internationalization, and packaging PyQt6 applications for distribution.
* **Case Studies and Practical Examples**: Throughout the book, we'll provide real-world examples and case studies to reinforce learning and demonstrate how PyQt6 can be used to solve practical problems.

By the end of this book, you'll have the knowledge and confidence to develop your own PyQt6 applications, whether you're building scientific tools, data analysis applications, or interactive visualizations.

## How to Use This Book

This book is designed to be accessible to readers of all levels, from beginners to experienced developers. Each chapter builds upon the concepts introduced in the previous chapters, gradually increasing in complexity. If you're new to Python or PyQt6, we recommend starting from the beginning and working your way through each chapter sequentially. However, if you're already familiar with the basics, feel free to jump to the chapters that interest you the most.

Throughout the book, you'll find code examples, explanations, and exercises to help reinforce your understanding of the material. We encourage you to follow along with the examples, experiment with the code, and apply what you've learned to your own projects.

## Let's Get Started!

Are you ready to embark on this exciting journey into the world of Python and PyQt6? Let's dive in and start exploring the possibilities together!

## Python basics and syntax

Python is a powerful programming language known for its simple syntax, making it easy to read and write. Python enables developers to write programs in fewer lines of code, increasing efficiency and reducing complexity. Let’s cover some of the fundamentals of Python.

------------------------------------------------------------------Updated

## How Python works?

Python is an interpreted language, means the code is executed line by line by the interpreter. Python interpreter reads the source code and executes it directly. Python interpreter reads your Python code and translates it into machine-readable bytecode. This bytecode is then executed by the Python Virtual Machine (PVM).

Python interpreter supports dynamic typing. It automatically determines the type of a variable based on the value assigned to it, and it manages memory allocation and deallocation using a built-in garbage collector. You can run Python interpreter in interactive mode, using that you can enter commands and expressions directly into the shell and see the results. Python comes with lot of standard libraries that provides a wide range of modules and functions for various tasks such as file I/O, networking, multimedia, and more.

## Python Installation:

Before we start on leaning python lets get python installed in our machine. Its good to read the book but also get our hand dirty by doing some sample programs for better understanding.

To install Python, you can download from below url. You can download the latest version and for specific OS like windows, macOs, Linux/UNIX or other. In case you need different version that will be also available in the same page. You can have more the one version of python installed in your machine.

<https://www.python.org/downloads/>

A screenshot of a computer

Description automatically generated

Once you download the python exe file, double click to run the installer. Before clicking **Install Now** select **Add** **python.exe to PATH.**

A screen shot of a computer

Description automatically generated

Once your installation is done you can verify the installed python by using python –version command.

**A screenshot of a computer

Description automatically generated**

To write Python program we should have your preferred text editor or integrated development environment (IDE), whether it's PyCharm, Visual Studio Code, or any other tool you prefer.

Let’s write our first program using python. We have created a Github repository for all the examples we are going to discuss in this book. You can refer in case you have doubt.

To run any program in a file, use the below command.

**python filename.py**

Create a file name FristProgram.py. Let’s print hello world! as first step.

print("Hello World!")

A screenshot of a computer

Description automatically generated

**Output:**

Hello World!

Here is the link for Github repo for FirstProgram.py [FirstProgram link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_1/FirstProgram.py)

**Variables and Datatypes:**

Variables serve as containers to store data values in Python. You assign a value to a variable using the assignment operator “=”. Variables can be declared either globally, accessible throughout the entire program, or locally, confined to a specific scope, such as within a function.

When naming variables, it’s important to follow some basic rules:

* + Variable names can contain letters, numbers and underscores.
  + Variable names must start with a letter or an underscore.
  + Variable names never start with a number and doesn’t contain whitespace.
  + Variable names are case-sensitive.
  + Avoid using Python keywords as variable names.

Here’s a list of common data types in Python along with examples:

In Python, we don’t explicitly declare the variable type. Instead declare a variable and assign a value to it.

**Integer:**  
x = 10  
print(type(x)) # <class 'int'>

**String:**  
y = 'Hello'  
print(type(y)) # <class 'str'>

**Float:**  
z = 5.25  
print(type(z)) # <class 'float'>

**Boolean:**  
isVisible = False  
print(type(isVisible)) # <class 'bool'>  
**Booleans can be either True or False**

**List:**  
myList = [1, 2, 3, 4]  
print(type(myList)) #<class 'list'>

**Tuple:**  
myTuple = (1, 2, 3, 4)  
print(type(myTuple)) #<class ' tuple'>

**Dictionary:**  
myDict = {'a': 1, 'b': 2}   
print(type(myDict)) #<class 'dict'>

You can find the sample for variable and data type in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_1/VariableAndDatatype.py).

**Casting:**

Casting in Python is a process of converting one data type into another. Python provides built-in functions to perform these conversions, allowing you to manipulate data of different types efficiently.

Below are some of the basic examples to change datatype.

x = str('3')  
print(x) # Output: 3  
y = float(3)  
print(y) # Output: 3.0  
  
z = bool(1)  
print(z) # Output: True

You can find the sample for type casting in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_1/TypeCasting.py).

**Comments:**

Writing comments in code is indeed a good habit. It helps not only yourself, but also other team members understand the functionality of the code. In Python, you can add comments by using the pound sign (#), and anything after the # is ignored by the interpreter. This allows you to add explanations, notes, or reminders within your code. Unlike some other programming languages, Python does not have a built-in syntax for multi-line comments using /\* … \*/.

# Comments  
X=5 # int  
Y=6 # string

**Conditional Statements:**

Condition statements are utilized for decision-making within a program. They evaluate a statement to determine whether it is true or false, and based on the result, a specific block of code is executed. In writing a conditional statement, we employ if, elif (not else if), and else. The elif and else conditional statements are optional. Furthermore, we can utilize nested if statements to construct more complex conditional statements.

age = 10  
if age > 5:  
 print("age is greater than 5")  
elif age < 5:  
 print("age is less than 5")  
else:  
 print("age is equal to 10")

You can find the sample for conditional statement in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_1/ConditionalStatement.py).

**Loops**

Loops serve to control flow structures and repetitively execute a block of code until a specified condition is met.

You have the flexibility to use either a for loop or a while loop. Within loops, you can skip iterations using **continue** statement, exit using the **break** statement, or proceed without any action using **pass** statement.

**# For Loop:**  
for i in range(3):  
 print(i)

**Output:**  
0  
1  
2

**#While Loop:**   
j=0  
while j < 3:  
 print(j)  
 j = j + 1

**Output:**  
0  
1  
2

You can find the sample for loop in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_1/Loop.py).

**Functions:**

Functions are reusable blocks of code designed to perform specific tasks. By writing functions, you can efficiently manage your codebase and easily reuse the same functionality in multiple sections of your program.

To define a function, you utilize the ‘def’ keyword followed by a unique name for the function. You can then call the function either with or without arguments, depending on its functionality.

# Without arguments

def sayHello():  
 print('Hello! ')  
sayHello() # Calling function without arguments

**Output:**  
Hello!

# With arguments  
def sayHelloByName(name):  
 print('Hello '+name +'!')  
sayHelloByName('Rob') # Calling function with arguments

**Output:**  
Hello Rob!

# Optional arguments  
def sayHelloByOptionalName(name = 'Kane'):  
 print('Hello '+name +'!')  
sayHelloByOptionalName() # Calling function by optional arguments

**Output:**  
Hello Kane!

## What is PyQt6?

PyQt6 serves as a collection of Python bindings for the Qt application framework, capable of running on all platforms supported by Qt, including Windows, macOS, Linux, and iOS. It is a powerful tool for developing graphical user interfaces (GUIs) using Python. PyQt6 mandates a minimum Python version of 3.x. Developed by Riverbank Computing, PyQt no longer supports PyQt4, with PyQt5 and PyQt6 being the current versions in use, with PyQt6 is the latest iteration.

PyQt6 is known for its ease of learning and utilization, offering a wide range of features and functionality. It helps developers to create rich user interfaces (UIs) incorporating widgets, toolbars, menus, layouts, signals, multimedia support, and beyond.

// Need to Verify

PyQt6 free versions are licensed under the GNU General Public License version 3. If your use of Riverbank's software is compatible with the GPL then you do not need to buy a commercial license.

If your use of Riverbank's software is not compatible with the GPL then you require a commercial license. There is no functional difference between the GPL versions and the commercial versions of Riverbank's software.

------------------------------------------------------------------Updated

## Installing PyQt6

Before proceeding with the installation of PyQt6, ensure that your system has Python version of 3.x. If you have not installed yet you can refer previous chapter Python Installation.

To install PyQt6, use below command:

pip install PyQt6

## Setting up your development environment

Getting Started with PyQt6

Let’s start on PyQt6, We can start from scratch for building a GUI application. Its better when we understand most of the features available in PyQt6. We have to leverage the list of features available to build a better version of application, which makes the UI look good at the same time user friendly and not make it complex. UI is always good to use when you keep it simple.

We can start creating simple application and widgets before creating a full-fledged UI application.

## Creating your first PyQt6 application

We can use QApplication and Qwidget to create our first application. Lets create a simple window with title Hello world.

from PyQt6.QtWidgets import QApplication, QWidget  
#Create an application  
app = QApplication([])  
#Create a window  
window = QWidget(windowTitle="Hello World!!")  
window.show()  
app.exec()

Output:

A screenshot of a computer

Description automatically generated

Will go through the code and explain what we are doing here to create our application.

**Step1:** We have to import Qpplication and Qwidget from PyQt6 QtWidgets.

**Step2:** Let’s create an application with empty array as arguments. We can use sys.args in case we provide any input.

**Step 3:** Create a window using Qwidget and set the title as Hello World!! And to show the window will call show() method.

**Step 4:** app.exec() method is called at last to execute the event loop where all the events are handled (Event loop will be taught in future chapter).

## Understanding PyQt6 widgets

PyQt6 is the basic for building GUI applications. Widget can hold other widgets together. widgets include buttons, labels, text boxes, menus, toolbars, and many others, allows to create a professional looking application. For widget you can set the height, width, styles, and more.

## Layout management in PyQt6

Layout management is most important part of creating graphical user interface because they control how widgets (such as buttons, labels, text fields, etc.) are positioned and resized within a window. When we specify the position and size of each widget manually then when we resize the window, position and size of widget won’t change based on the size of window. If we open the application in different resolutions same issue will happen and layout will be distorted. To solve this issue, we can use default layouts available in PyQt6. When we add more widget and want to do alignment, we can use below four layouts to position the widgets, each has its unique behaviour.

* **QHBoxLayout** - Align widgets horizontally.
* **QVBoxLayout**  - Align widgets Vertically.
* **QGridLayout** - Align widgets by positioning in grid structure.
* **QStackedLayout** - Add widgets in Stack in front of one another.

**QHBoxLayout:**

## QHBoxLayout class arranges widgets horizontally. When you add new widgets, each will be placed next to each other. QHBoxLayout class inherits from QBoxLayout, which further inherits from QLayout. This inheritance allows to make effective layout management in a GUI.

## Let’s create a sample application to see how widgets are placed when we use QHBoxLayout. To use QHBoxLayout we need to import the class from PyQt6.QtWidgets.

from PyQt6.QtWidgets import QApplication, QWidget, QHBoxLayout, QPushButton  
  
# Initialize the application  
app = QApplication([])  
  
# Create the main window  
window = QWidget()  
window.setWindowTitle('QHBoxLayout')  
window.setMinimumWidth(300)  
window.setMinimumHeight(300)  
  
# Create a QHBoxLayout  
layout = QHBoxLayout()  
  
# Create widgets   
button1 = QPushButton('Button 1')  
button2 = QPushButton('Button 2')  
button3 = QPushButton('Button 3')  
button4 = QPushButton('Button 4')  
  
# Add widgets to Layout  
layout.addWidget(button1)  
layout.addWidget(button2)  
layout.addWidget(button3)  
layout.addWidget(button4)  
  
# Set the layout  
window.setLayout(layout)  
  
# Show the window  
window.show()  
  
# Run the application's event loop  
app.exec()

## **Output:**

**A screenshot of a computer

Description automatically generated**

You can find the sample for horizontal layout in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_2/HorizontalLayout.py).

In above pic you can see the buttons are arranged horizontally. First, we have to define a QHBoxLayout (layout = QHBoxLayout()) and then create more than one widgets in our case we have created widget using QPushButton. Once the widgets are created, we need add all the widgets one by one into the layout using layout.add(widgetName) method. We need to add widgets in the order that must be displayed. Finally for the window we need to set the layout using window.setLayout(layoutName) method.

You can use below methods in layout for positioning inside layout.

|  |  |
| --- | --- |
| setSpacing() | To set space between widgets |
| addStretch() | To push a widget from another |
| setContentsMargins() | To set margins around the edges of layout |

## **QVBoxLayout:**

## QVBoxLayout class arranges widgets vertically. When you add new widgets, each will be placed from top to bottom. Like QHBoxLayout, QVBoxLayout also inherits from QBoxLayout class, which further inherits from QLayout. This inheritance allows to make effective layout management in a GUI.

## Let’s create a sample application to see how widgets are placed when we use QVBoxLayout. To use QVBoxLayout we need to import the class from PyQt6.QtWidgets.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton  
  
# Initialize the PyQt application  
app = QApplication([])  
  
# Create the main window  
mainWindow = QWidget()  
mainWindow.setWindowTitle("QVBoxLayout")  
mainWindow.setMinimumWidth(300)  
mainWindow.setMinimumHeight(300)  
# Create a QVBoxLayout instance  
layout = QVBoxLayout()  
  
# Create and add widgets to the layout in one line  
layout.addWidget(QPushButton("Button 1"))  
layout.addWidget(QPushButton("Button 2"))  
layout.addWidget(QPushButton("Button 3"))  
layout.addWidget(QPushButton("Button 4"))  
  
# Set the layout   
mainWindow.setLayout(layout)  
  
# Display the main window  
mainWindow.show()   
  
# Start the event loop  
app.exec()

**Output:**

## A screenshot of a computer Description automatically generated

You can find the sample for vertical layout in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_2/VerticalLayout.py).

In above pic you can see the buttons are arranged vertically one by one. First, we need to define a QVBoxLayout (layout = QVBoxLayout()) and then create QPushButton and add it to layout widget using layout using layout.add(widgetName) method. We need to add widgets in the order that must be displayed. Finally for the window we need to set the layout using window.setLayout(layoutName) method.

**QGridLayout:**

QGridLayout is little bit different than both QHBoxLayout and QVBoxLayout. QGridLayout allows widgets to be arranged in a grid format. QGridLayout organizes widgets in a two-dimensional grid, where each cell in the grid can contain one widget. The grid layout can be used for creating forms or any application requiring a structured layout. QGridLayout class inherits from QLayout class.

## Let’s create a sample application to see how widgets can be placed using QGridLayout. To use QGridLayout we need to import the class from PyQt6.QtWidgets.

from PyQt6.QtWidgets import QApplication, QWidget, QGridLayout, QPushButton  
  
# Initialize the application  
app = QApplication([])  
  
# Create a window  
window = QWidget()  
window.setWindowTitle('QGridLayout')  
window.setMinimumWidth(600)  
window.setMinimumHeight(600)  
  
# Create a QGridLayout  
layout = QGridLayout()  
  
# Create widgets   
button1 = QPushButton('Button 1')  
button2 = QPushButton('Button 2')  
button3 = QPushButton('Button 3')  
button4 = QPushButton('Button 4')  
button5 = QPushButton('Button 5')  
button6 = QPushButton('Button 6')  
button7 = QPushButton('Button 7')  
  
  
# Add widgets to the layout at specified positions  
layout.addWidget(button1, 0, 0) # Arguments row, column  
layout.addWidget(button2, 0, 1, 1, 2) # Arguments row, column, rowSpan, columnSpan  
layout.addWidget(button3, 0, 3) # Arguments row, column  
layout.addWidget(button4, 1, 0, 1, 2) # Arguments row, column, rowSpan, columnSpan  
layout.addWidget(button5, 1, 2) # Arguments row, column  
layout.addWidget(button6, 1, 3) # Arguments row, column  
layout.addWidget(button7, 2, 0, 1, 4) # Arguments row, column, rowSpan, columnSpan  
  
# Apply the layout to the window  
window.setLayout(layout)  
  
# Display the window  
window.show()  
  
# Start the application's event loop  
app.exec()

## **Output:**

## A screenshot of a computer Description automatically generated

You can find the sample for grid layout in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_2/GridLayout.py).

In above pic you can see the buttons are positioned in Grid layout. First, we need define a QGridLayout (layout = QGridLayout()) and then create widgets in our case we have created widget using QPushButton. Once the widgets are created, we need add all the widgets one by one into the layout using layout.add(widgetName, row, column) or layout.add(widgetName, row, column, rowSpan, columnSpan) method. We need to add widgets based on row and column and also, we can specify the row span and column span to make widget size to multiple cells. Finally for the window we need to set the layout using window.setLayout(layoutName) method.

**QStackedLayout:**

**QStackedLayout** allows you to stack multiple widgets on top of one another. Only one widget will be visible at a time. Stacked layout behaves same as a tab widget. Stack layout is useful in creating multiple pages and show one page at a time. You can change the page when there are user interactions.

You can use below methods in stack layout.

|  |  |
| --- | --- |
| setCurrentIndex() | To change the widget visibility |
| insertWidget () | To add widget into stack |
| removeWidget () | To remove widget from stack |

Let’s create an application where we can add widgets in stacked layout and switch between widgets on button click.

from PyQt6.QtWidgets import QApplication, QWidget, QStackedLayout, QPushButton, QVBoxLayout  
  
# Initialize the application  
app = QApplication([])  
  
# Create a window  
window = QWidget()   
window.setWindowTitle('Stacked Layout')  
window.setMinimumWidth(600)  
window.setMinimumHeight(500)  
  
# Create a QVBoxLayout to hold the QStackedLayout and a Switch button  
mainLayout = QVBoxLayout()  
  
# Create the QStackedLayout  
stackedLayout = QStackedLayout()  
  
# Create widgets  
widget1 = QPushButton('Page 1')  
widget1.setStyleSheet('background-color: red;')  
widget2 = QPushButton('Page 2')  
widget2.setStyleSheet('background-color: green;')  
widget3 = QPushButton('Page 3')  
widget3.setStyleSheet('background-color: blue;')  
  
# Add widgets to stacked layout  
stackedLayout.addWidget(widget1)  
stackedLayout.addWidget(widget2)  
stackedLayout.addWidget(widget3)  
  
totalViews = 3  
# Function to switch between views  
def onSwitchView():  
 currentIndex = stackedLayout.currentIndex()  
 if currentIndex < totalViews - 1:  
 stackedLayout.setCurrentIndex(currentIndex + 1)  
 else:   
 stackedLayout.setCurrentIndex(0)  
  
# Button to switch views  
switchButton = QPushButton('Switch View')  
switchButton.clicked.connect(onSwitchView)  
  
# Add layout and button to main layout  
mainLayout.addLayout(stackedLayout)  
mainLayout.addWidget(switchButton)  
  
# Set main layout to window  
window.setLayout(mainLayout)  
  
# Show window  
window.show()  
  
# Start event loop  
app.exec()

**Output:**

**A screenshot of a computer

Description automatically generated**

You can find the sample for stack layout in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_2/StackLayout.py).

**Exercises:**

Let’s start doing some coding by creating sample applications to implement different layouts.

1. Create a login form using QHBoxLayout and QVBoxLayout.

2. Create a Calculator using QGridLayout, QHBoxLayout and QVBoxLayout.

3. Create an application which behaves like a tab widget using QStackedLayout and Buttons.

## Event handling in PyQt6

Events is most important part of UI application. Events are generated when either user does any interactions in UI or by system events. Mouse click, key press, drag, and more are some of the user generated events. System generated events can be data arriving in network, timers, etc. When ever the application is executed, the main loop runs which handles the events and sends to the respective widgets.

To understand more about events lets go through the example of mouse click event. When a user clicks on a widgets QMouseEvent is generated. Each event generated is represented as event object which will hold all the information related to the event for example which key pressed in keyboard or mouse position. The generated event is handled by the event handler using a function where we can do the custom functionality.

Let’s create a program to see how we can trigger a user event and handle that event using an event handler.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton  
  
class MyApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setMinimumWidth(400)  
 self.setMinimumHeight(400)  
   
 def initUI(self):  
 self.count = 0  
 layout = QVBoxLayout()  
 # Create a button  
 self.button = QPushButton('Click', self)  
   
 # Create Event handler  
 self.button.clicked.connect(self.onButtonClick)  
  
 layout.addWidget(self.button)  
 layout.setContentsMargins(100,100,100,100)  
 self.setLayout(layout)  
  
 # Event Handler function  
 def onButtonClick(self):  
 self.count = self.count + 1  
 self.button.setText('Clicked '+ str(self.count))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Initialize the PyQt application   
 app = QApplication([])  
 # Create a window  
 w = MyApp()  
 # Display the window  
 w.show()  
 # Start the event loop  
 app.exec()

**Output:**

A screenshot of a computer

Description automatically generated

You can find the sample for stack layout in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_2/UserEvent.py).

**Event Propagation:**

When an event triggered like mouse events, key board events, or custom events, the events is propagated from child to parents. Event can be either handled in child or passed to parents. Event starting from child and moving up to parents is called event bubbling. When a child widget accepts the event using **event.accept()** method then further event propagation is stopped and handle the event your way. If the event is ignored, then it’s propagated to its parents.

Let’s write an example to see how event propagation handled is handled with **accept()** method.

import sys  
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QLabel  
  
class ChildWidget(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setLayout(QVBoxLayout())  
 self.label = QLabel('Click on Child widget.')  
 self.layout().addWidget(self.label)  
  
 def mousePressEvent(self, event):  
 print('Sub Child Event triggered.')  
 # Accept the event  
 event.accept()  
  
class ParentWidget(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setLayout(QVBoxLayout())  
 self.subChild = ChildWidget()  
 self.layout().addWidget(self.subChild)  
 self.setGeometry(400,200,400,400)  
  
 def mousePressEvent(self, event):  
 print('Event Reached Parent')  
   
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication(sys.argv)  
 w = ParentWidget()  
 w.show()   
  
 app.exec()

**Output:**

Sub Child Event triggered.

**Note:**

In parent widget mouse press event is handled but the event didn’t reach because in child we handled in our away and stopped propagation by using **accept()** method

You can find the sample Even handling program in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_2/EventPropagation_Accept.py).

**Exercises:**

Let’s do an exercise for Events.

1. Write a program for ignoring event using **ignore()** method and propagating parent. You can refer the accept method example.
2. Create a system event using timer and show a label with counter time in seconds and stop the counter when it reaches 20 second using the event.

PyQt6 Widgets in Depth

PyQt6 offers wide range of widgets that can be used to create a robust and professional applications. When there are more widgets available in GUI library that will help us to give more user-friendly application for the user and reduce our time in creating custom widgets. One of the good things about PyQt widgets are we can use the existing widget and customize in case we need some custom functionality.

## Commonly used widgets (buttons, labels, text boxes, etc.)

We can go through some of the common widgets widely used in PyQt6.

* + Button
  + Label
  + Textbox
  + Text Editor
  + ComboBox
  + CheckBox
  + RadioButton
  + Slider

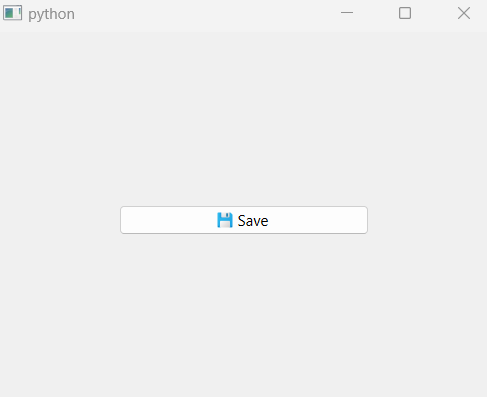
**Button:**

Button is one of the commonly used widgets in UI application. Buttons are used to interact with application. QPushButton is used to create the button. Button can display text, icon, or both.

Let’s create a button with icon, text and add click event functionality using PyQt6.

import os  
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton  
from PyQt6.QtGui import QIcon  
  
# local machine repository path  
path = os.environ.get('SampleProgram')  
  
class ButtonApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
   
 # Create a button  
 self.button = QPushButton('Save', self)  
   
 # Create icon with image page  
 icon = QIcon(path + 'assets/images/save\_icon.png')  
 # Set icon to button  
 self.button.setIcon(icon)  
  
 layout.addWidget(self.button)  
 layout.setContentsMargins(100, 100, 100, 100)  
 # Create Event handler  
 self.button.clicked.connect(self.onButtonClick)  
 self.setLayout(layout)  
  
 # Event Handler function  
 def onButtonClick(self):  
 print('Save Button clicked!')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = ButtonApp()  
 w.show()  
 app.exec()

**Output:**



You can find the sample for button in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_3/Button.py).

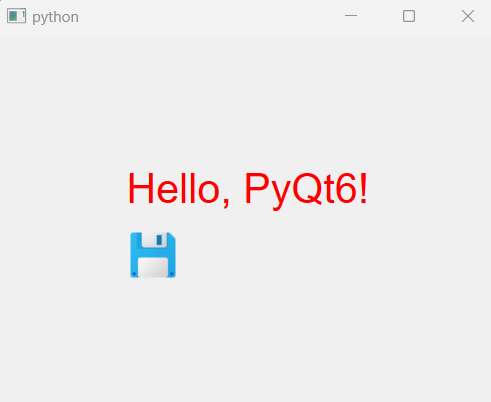
**Label:**

QLabel is simple and one of the most frequently used widgets in PyQt6 application. When we create a form, QLabel is the important widget going to be used. Label can be single or multi line text. QLabel is used to add description, instructions, or image. In Qlable we can use **setOpenExternalLinks(True)** methodto display hyperlinks and on click of link we can open the url in the default web browser. In QLabel you can change font type, size, color, and more to make styling better. You can align QLabel respective to another widget.

Let’s create two QLabel widget using text and image. Will try out some styling properties for QLabel.

import os  
from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QLabel   
from PyQt6.QtGui import QFont, QPixmap  
from PyQt6.QtCore import Qt  
  
# local machine repository path  
path = os.environ.get('SampleProgram')  
  
class LabelApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
   
 label = QLabel('Hello, PyQt6!')  
 font = QFont()  
 font.setFamily("Arial")  
 font.setPointSize(25)  
 label.setFont(font)  
   
 # Set the color, You can use stylesheet to set font also   
 label.setStyleSheet("color: red")  
 label.setAlignment(Qt.AlignmentFlag.AlignCenter)  
 layout.addWidget(label)  
  
 imageLabel = QLabel()  
 # Load the image  
 pixmap = QPixmap(path + 'assets/images/save\_icon.png')  
 # Set the image in the QLabel widget  
 imageLabel.setPixmap(pixmap)  
 layout.addWidget(imageLabel)  
  
 layout.setContentsMargins(100, 100, 100, 100)  
 self.setLayout(layout)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = LabelApp()  
 w.show()  
 app.exec()

**Output:**



You can find the sample for button in [GitHub Link](https://github.com/TenetTechnetronics/Building_Engineering_Applications_With_Python/blob/main/Programs/Chapter_3/Label.py).

**Textbox:**

QLineEdit widget in PyQt6 applications is used for obtaining input from the user in the form of single-line text fields. We can use the text box to enter any data like email, phone number and much more. We can set placeholder text to give more information about the field user have to input. We can validate and format the user entered input in the text box. You can add custom validation logic in case you need. QLineEdit can be set as read-only when you don’t want user to update the existing value in the textbox.

|  |  |
| --- | --- |
| setText() | To set a text in line edit |
| setMaxLength() | To set maximum number of characters allowed. |
| textChanged | Triggered when text change |
| returnPressed | Triggered when enter pressed |

Here is the basic example for QLineEdit, using that we can get the value entered by the user and handle when text changed or return pressed.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QLabel, QLineEdit  
  
class TextboxApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
 self.setWindowTitle('QLineEdit Example')  
   
 def initUI(self):   
 # Create a layout  
 layout = QVBoxLayout()  
  
 # Create a QLineEdit  
 self.lineEdit = QLineEdit()  
 self.lineEdit.setPlaceholderText('Enter your text here')  
 # Text changed  
 self.lineEdit.textChanged.connect(self.updateValue)  
 # Return Pressed  
 self.lineEdit.returnPressed.connect(self.returnText)  
 # set maximum length user can enter in textbox  
 self.lineEdit.setMaxLength(10)  
  
 # Add LineEdit  
 layout.addWidget(self.lineEdit)  
  
 self.label = QLabel('Textbox Value')  
   
 # Add Label  
 layout.addWidget(self.label)  
  
 self.setLayout(layout)  
   
 def updateValue(self):  
 self.label.setText(self.lineEdit.text())  
  
 def returnText(self):  
 print('Texbox value when return pressed - ' + self.lineEdit.text())  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = TextboxApp()  
 w.show()  
 app.exec()

**Output:**

A screenshot of a computer

Description automatically generated

**Text Editor:**

QTextEdit provides a powerful, flexible editor that supports both plain and rich text formatting. QLineEdit is used for single line input but QTextEdit is used for multi-line input. QTextEdit supports both plain text and rich text, you can add html content in the text editor. In QTextEdit you can change fonts, colors size to a paticular line or whole text, you can insert text too. In text editor you can do all clipboard functionalities like copy, paste, undo, redo, and more. We can set read-only mode for text editor. QTextEdit supports vertical and horizontal scrolls.

|  |  |
| --- | --- |
| setStyleSheet() | To style the text in text editor |
| setPlainText () | To set plain text value in text editor |
| setHtml () | To set HTML text value in text editor |
| toPlainText () | To return value as plain |
| toHtml() | To return value as HTML |

Let’s write a program to see how to use QTextEdit and its functions.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QTextEdit  
  
class TextEditorApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
 self.setWindowTitle('QTextEdit Example')  
   
 def initUI(self):   
 # Create a layout  
 layout = QVBoxLayout()  
 # Create a QTextEdit  
 self.textEdit = QTextEdit()  
 self.textEdit.setHtml('<h1>PyQt6</h1><p>This is a <b>rich text</b> editor.</p>')  
  
 # Add TextEdit  
 layout.addWidget(self.textEdit)  
 button = QPushButton('Display Value')  
 button.clicked.connect(self.displayValue)  
 layout.addWidget(button)  
  
 self.setLayout(layout)  
   
 def displayValue(self):  
 print('Plain text')  
 print(self.textEdit.toPlainText())  
 print('Rich text')  
 print(self.textEdit.toHtml())  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = TextEditorApp()  
 w.show()  
 app.exec()

**Output:**

A screenshot of a computer

Description automatically generated

**ComboBox:**

QComboBox widget component combines a button or editable field with a dropdown list, allowing users to select from a list of options or enter their own choice. It's one of the way to present a selection list to the user. QComboBox one of the come use case is giving the list of states in a couty for user to select when the enter address. Instead of typing the value you can provide user with a predefined list from that they can choose. QComboBox allow users to enter the input rather than selecting from the list, making it flexible for user.

Some of the common methods used in QComboBox are listed below.

|  |  |
| --- | --- |
| setEditable(True) | Set the ComboBox editable |
| addItems([item1, item2, ...]) | To add one or more number of items to the list. |
| addItem(item) | To add an item into the list. |
| removeItem(index) | To remove an item from the list. |
| count() | Returns the number of items in list. |
| currentText() | Returns the currently selected item. |
| currentIndex() | Returns the currently selected item index. |
| setCurrentIndex(index) | Set the index as selected item. |
| setStyleSheet | Set styling properties. |
| placeholderText | To set the place holder text. |
| currentTextChanged | Triggered whenever current text changed. |
| currentItemChanged | Triggered whenever current item changed. |

## 

Let’s write a program to create Combobox and see how to use some the functions available in Combobox.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QComboBox  
  
class ComboBoxApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
 self.setWindowTitle('QComboBox Example')  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
   
 # Create a QComboBox  
 self.comboBox = QComboBox()  
 # Add multiple items to the QComboBox  
 self.comboBox.addItems(['Option 1', 'Option 2', 'Option 3', 'Option 4'])  
 # Add single item  
 self.comboBox.addItem('Option 5', userData="This is Option 5")  
 self.comboBox.setEditable(True)  
  
 # Add combobox  
 layout.addWidget(self.comboBox)  
  
 # Trigger on Index changed  
 self.comboBox.currentIndexChanged.connect(self.onSelectedIndexChanged)  
 # Trigger on Text changed  
 self.comboBox.currentTextChanged.connect(self.onTextChanged)  
  
 # Add button to remove item from list  
 button = QPushButton('Remove Item')  
   
 # Remove item on Click  
 button.clicked.connect(self.removeItem)  
 layout.addWidget(button)  
  
 self.setLayout(layout)  
   
 def onSelectedIndexChanged(self, index):  
 print('item ', index, self.comboBox.currentIndex())  
 print('Selected Text', self.comboBox.currentText())  
  
 def onTextChanged(self, itemText):  
 print('onTextChanged ', itemText)  
  
 def removeItem(self):  
 print('Before Remove Item count ', self.comboBox.count())  
 self.comboBox.removeItem(2)  
 print('After Remove Item count ', self.comboBox.count())  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = ComboBoxApp()  
 w.show()  
 app.exec()

**Output:**

A screenshot of a computer

Description automatically generated

**CheckBox:**

PyQt6 QCheckBox widget is used to allow the user to either checked or unchecked. It offers user a simple way to collect input for binary choices, such as toggling options on or off. QCheckBox not only has True or False option. There is another option called TriState mode where checkbox state is not selected or unselected, its a intermediate state called partially checked. Tristate is used in complex forms where the answer is not a straght forward one. In Tristate when you select a checkbox first it will set as partially selected on second click checkbox will be fully selected. To enable tristate for checkbox you should use setTriState(True) method.

|  |  |
| --- | --- |
| setTriState() | To set tri state for checkbox |
| setChecked() | To set True or False value in checkbox |
| setCheckState() | To set state of checkbox |
| stateChanged | Trigger signal when state changed |

Lets write a example program to see how we can add a checkbox and also use Tristate for more understanding.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QCheckBox, QLabel  
from PyQt6.QtCore import Qt  
  
class CheckBoxApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
 self.setWindowTitle('QCheckBox Example')  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
  
 # Create a QCheckBox  
 self.checkbox = QCheckBox('My CheckBox')  
 # To set Tristate  
 self.checkbox.setTristate(True)  
 # stateChanged Event  
 self.checkbox.stateChanged.connect(self.onStageChanged)  
   
 # Add label to show state and value of checkbox  
 self.label = QLabel()  
 # set stylesheet  
 self.label.setStyleSheet("font-size: 15px; font-weight: bold; color: Red;")  
 layout.addWidget(self.checkbox, alignment=Qt.AlignmentFlag.AlignCenter)  
 layout.addWidget(self.label, alignment=Qt.AlignmentFlag.AlignCenter)  
 self.setLayout(layout)  
  
 # Set default value for checkbox   
 self.checkbox.setCheckState(Qt.CheckState.Checked)  
   
 def onStageChanged(self, value):  
 self.label.setText('Checkbox State: '+ str(self.checkbox.checkState()) + ', Value: '+ str(value))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = CheckBoxApp()  
 w.show()  
 app.exec()

**Output:**

A screenshot of a computer

Description automatically generated

**RadioButton:**

PyQt6 QRadioButton widget gives users to select a single option from multiple options available. In a QRadioButton group each radio button is mutually exclusive, means selecting one option will automatically deselect the previously selected option. In QCheckBox you can select multiple options in a group of checkboxes but in radio buttons a user must choose one option from the give list of options. When you want to group a list of radio buttons **QButtonGroup** can be used. In an application you have more than one group of radio buttons.

|  |  |
| --- | --- |
| setChecked() | To set radio button selected using boolean |
| setDisabled() | To disable radio button |
| Toggled() | Trigger signal when radio button toggled |

Let’s write a program and see how to use radio buttons and some of the methods available in it.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QRadioButton, QLabel, QButtonGroup  
  
class RadioButtonApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 200)  
 self.setWindowTitle('QRadioButton Example')  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
  
 # Create a QRadioButton  
 radio1 = QRadioButton('Option 1')  
 radio2 = QRadioButton('Option 2')  
 radio3 = QRadioButton('Option 3')  
   
 radio1.toggled.connect(self.onChange)  
 radio2.toggled.connect(self.onChange)  
 radio3.toggled.connect(self.onChange)  
  
 # Do stop user from selecting option  
 #radio3.setCheckable(False)  
 # To disable radio button  
 #radio2.setDisabled(True)  
   
 # Add Radio button  
 layout.addWidget(radio1)  
 layout.addWidget(radio2)  
 layout.addWidget(radio3)  
  
 # Create a button group for the radio buttons  
 self.button\_group = QButtonGroup()  
 self.button\_group.addButton(radio1)  
 self.button\_group.addButton(radio2)  
 self.button\_group.addButton(radio3)  
  
  
 # Add label to show state and value of checkbox  
 self.label = QLabel()  
 # set stylesheet  
 self.label.setStyleSheet("font-size: 15px; font-weight: bold; color: Red;")  
 layout.addWidget(self.label)  
 self.setLayout(layout)  
   
 # Set second option as selected by default  
 radio2.setChecked(True)  
  
   
 def onChange(self, value):  
 print('Radio button: ', self.sender().text(), value)  
  
 # Update selected radio button name and value in label when selected   
 if value == True:  
 self.label.setText('Radio button: '+ self.sender().text() + ', Value: '+ str(value))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = RadioButtonApp()  
 w.show()  
 app.exec()

**Output:**

**A screenshot of a computer

Description automatically generated**

## Advanced widgets (tables, trees, tab, dialogs, etc.)

## 

PyQT6 offers advanced widgets that we can use in our application to meet complex requirements. We can see some of the advanced widget available in PyQt6.

**Table:**

PyQt6 QTableWidget/QTableView is used for creating and managing tables. Table allows you to add, remove or modify data. User can add text, icons, checkbox in table. Using Table you can select, sort, or filter the data. You can customize table's appearance and behaviour to make UI good.

**QTableWidget Vs QTableView:**

QTableWidget is used for simple table where you dont need more customization. In you want to disbale a large data set without any modifications QTableWidget is perfect choice. And it's easy to start with the default data model.

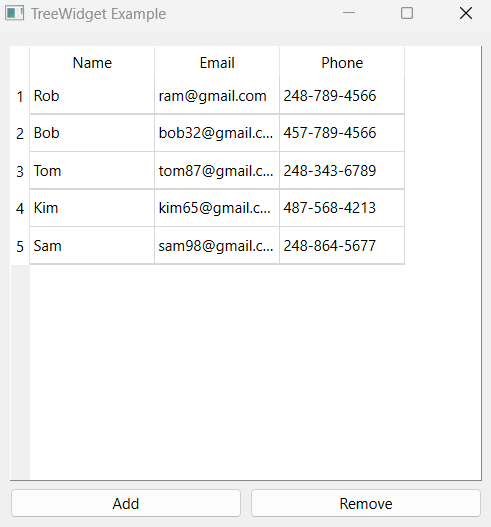
QTableView is used for complex tables that needs to be customized. If you use QTableView then need to create your own data model. it might take more time comapred to QTableWidget but you can customize the functionality.

|  |  |
| --- | --- |
| setColumnCount() | To set the column count |
| setRowCount() | To set the row count |
| setHorizontalHeaderLabels() | To set horizontal header labels for the table |
| setItem() | To set item at a given row and column index |
| insertRow() | Insert a row at the give index |
| removeRow() | Remove a row from the table |
| rowCount() | Returns the row count |
| currentRow() | Returns the current row index |

Let’s write a program and understand how to use the QTableWidget in PyQt6.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QTableWidget, QTableWidgetItem, QPushButton  
  
class TableWidgetApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 400)  
 self.setWindowTitle('TableWidget Example')  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
   
 self.table = QTableWidget(5, 3)  
 self.table.setHorizontalHeaderLabels(["Name", "Email", "Phone"])  
   
 # Data for Table  
 data = [  
 {"name":"Rob", "email":"ram@gmail.com", "phone": "248-789-4566"},   
 {"name":"Bob", "email":"bob32@gmail.com", "phone": "457-789-4566"},   
 {"name":"Tom", "email":"tom87@gmail.com", "phone": "248-343-6789"},  
 {"name":"Kim", "email":"kim65@gmail.com", "phone": "487-568-4213"},  
 {"name":"Sam", "email":"sam98@gmail.com", "phone": "248-864-5677"}  
 ]  
  
 # SetItem to table  
 for index, item in enumerate(data):  
 self.table.setItem(index, 0, QTableWidgetItem(item['name']))  
 self.table.setItem(index, 1, QTableWidgetItem(item['email']))  
 self.table.setItem(index, 2, QTableWidgetItem(item['phone']))  
  
 layout.addWidget(self.table)  
 hLayout = QHBoxLayout()  
 button1 = QPushButton("Add")  
 button2 = QPushButton("Remove")  
 button1.clicked.connect(self.addItem)  
 button2.clicked.connect(self.removeItem)  
 hLayout.addWidget(button1)  
 hLayout.addWidget(button2)  
  
 layout.addLayout(hLayout)  
 self.setLayout(layout)  
  
 def addItem(self):  
 index = self.table.rowCount()  
 self.table.insertRow(index)  
 self.table.setItem(index, 0, QTableWidgetItem('name' + str(index)))  
 self.table.setItem(index, 1, QTableWidgetItem('email' + str(index)))  
 self.table.setItem(index, 2, QTableWidgetItem('phone' + str(index)))  
   
 def removeItem(self):  
 self.table.removeRow(self.table.currentRow())  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = TableWidgetApp()  
 w.show()  
 app.exec()

**Output:**



**TreeView:**

PyQt6 QTreeWidget allows for the display of hierarchical data in a tree structure. It is useful for applications that require the visualization of complex data relationships in a user-friendly format. When you want to display the items in parent child relationship structure you can use QTreeWidget. When we create a QTreeWidget we need to set the columns count, header labels. Then you can populate the tree widget using QTreeWidgetItem. You can customize tree view and add custom login in case need like adding checkbox to tree node.

|  |  |
| --- | --- |
| setColumnCount() | To set the number of columns |
| setHeaderLabels() | To set header labels for the columns |
| expandl() | To expand a node |
| expandAl() | To expand all node |
| collapse() | To collapse a node |
| collapseAll() | To collapse all node |
| sortByColumn() | To sort tree based on column data |
| setCheckState() | Add checkbox to tree node for selection |
| setText(column, value) | Add text in the given column |

Write a program to see how we can create a QTreeWidget and use above methods in it.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QTreeWidget, QTreeWidgetItem, QPushButton  
from PyQt6.QtCore import Qt  
  
class TreeWidgetApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 400)  
 self.setWindowTitle('TreeWidget Example')  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
   
 # Create tree widget  
 self.tree = QTreeWidget()  
  
 # Set number of columns  
 self.tree.setColumnCount(2)  
 # Set column headers  
 self.tree.setHeaderLabels(["Car Company", "Car Model"])  
  
 # Data for tree  
 data = {  
 "Honda": ["Accord", "Pilot", "CR-V"],  
 "Toyota": ["Camry", "Rav4"],  
 "Hyundai": ["Accent"],  
 "Subaru": ["Outback", "Crosstrek", "Forester"]  
 }  
  
 # Populate tree widget with data  
 for company, carModel in data.items():  
 companyItem = QTreeWidgetItem(self.tree)  
 companyItem.setText(0, company)  
   
 companyItem.setCheckState(0, Qt.CheckState.Checked)  
   
 for model in carModel:  
 modelItem = QTreeWidgetItem(companyItem)  
 modelItem.setText(1, model)  
  
 # Show the tree widget  
 self.tree.show()  
 self.tree.sortByColumn(1, Qt.SortOrder.AscendingOrder)  
 layout.addWidget(self.tree)  
  
 hLayout = QHBoxLayout()  
 button1 = QPushButton("Expand")  
 button2 = QPushButton("Collapse")  
 button1.clicked.connect(self.expand)  
 button2.clicked.connect(self.collapse)  
 hLayout.addWidget(button1)  
 hLayout.addWidget(button2)  
  
 layout.addLayout(hLayout)  
 self.setLayout(layout)  
  
 def expand(self):  
 self.tree.expandAll()  
   
 def collapse(self):  
 self.tree.collapseAll()  
   
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = TreeWidgetApp()  
 w.show()  
 app.exec()

**Output:**

A screenshot of a computer

Description automatically generated

Tab widget:

PyQt6 QTabWidget that allows users to create tabbed interfaces for their applications. This widget allows you to organize the content into separate tabs, so it will be easier for users to navigate through different tabs and work on the content. Tabs can be easily added, moved, and removed. in Tab you can keep the tile as text, icon, or both. You can change the tab bar orientation to keep it in left, top, right, or bottom.

|  |  |
| --- | --- |
| addTab() | To add a tab in QTabwidget |
| removeTab() | To remove a tab |
| setTabIcon() | To set tab icon |
| setTabShape() | To set tab shape |
| insertTab() | To insert new tab at index |
| removeTab() | To remove a tab |
| count() | Return number of tabs |
| setTabEnabled() | To enable/disable tab |
| setTabText() | To set/update tab text |
| setMovable() | To enable or disable tab rearrange |
| setTabPosition() | To set tab location (left, top, right, or bottom) |
| currentIndex() | Return current index of tab |

Let’s see how to create QTabWidget by adding, removing tab, insert tab at a given index.

import os  
from PyQt6.QtWidgets import QApplication, QMainWindow, QWidget, QTabWidget, QHBoxLayout, QVBoxLayout, QLabel, QPushButton  
from PyQt6.QtGui import QIcon  
from PyQt6.QtCore import Qt  
  
# local machine repository path  
path = os.environ.get('SampleProgram')  
  
class TabWidgetApp(QMainWindow):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setGeometry(100, 100, 400, 400)  
 self.setWindowTitle('TabWidget Example')  
 self.initUI()  
   
 def initUI(self):  
 # Create Tab widget  
 self.tabWidget = QTabWidget()  
 self.addTabs()  
   
 # Enable Tab rearrange  
 self.tabWidget.setMovable(True)  
 self.setCentralWidget(self.tabWidget)  
   
 def addTabs(self):  
 tab1 = QWidget()  
 layout1 = QVBoxLayout()  
 label1 = QLabel('This is the content of Tab 1.')  
 layout1.addWidget(label1)  
 tab1.setLayout(layout1)  
  
 button1 = QPushButton("Add")  
 button2 = QPushButton("Remove")  
 button1.clicked.connect(self.addTab)  
 button2.clicked.connect(self.removeTab)  
 layout1.addWidget(button1)  
 layout1.addWidget(button2)  
  
 # Add tab to the tab widget  
 self.tabWidget.addTab(tab1, 'Tab 1')  
   
 tab2 = QWidget()  
 layout2 = QVBoxLayout()  
 label2 = QLabel('This is the content of Tab 2.')  
 layout2.addWidget(label2)  
 tab2.setLayout(layout2)  
 # Tab icon is optional.  
 # Create icon with image page  
 self.icon = QIcon(path + 'assets/images/save\_icon.png')  
   
 # Add tab to the tab widget  
 self.tabWidget.addTab(tab2, 'Tab 2')  
 self.tabWidget.setTabIcon(1, self.icon)  
  
 tab3 = QWidget()  
 layout3 = QVBoxLayout()  
 label3 = QLabel('This is the content of Tab 3.')  
 layout3.addWidget(label3)  
 tab3.setLayout(layout3)  
   
 # Add tab to the tab widget  
 self.tabWidget.addTab(tab3, 'Tab 3')  
 # Set icon for Tab  
 self.tabWidget.setTabIcon(2, self.icon)  
  
 def addTab(self):  
 tabCount = self.tabWidget.count()  
 tab = QWidget()  
 layout = QVBoxLayout()  
 label = QLabel('This is the content of Tab ' + str(tabCount + 1))  
 layout.addWidget(label)  
 tab.setLayout(layout)  
 # Add Tab at index  
 self.tabWidget.insertTab(tabCount, tab, 'Tab ' + str(tabCount + 1))  
 self.tabWidget.setTabIcon(tabCount, self.icon)  
  
 def removeTab(self):  
 if self.tabWidget.count() > 1:   
 # Remove Tab at index  
 self.tabWidget.removeTab(1)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = TabWidgetApp()  
 w.show()  
 app.exec()

**Output:**

A screenshot of a computer

Description automatically generated

**Dialog:**

PyQt6 Dialogs components are used for interacting with users, dialogs are used for decision making, display information, or request input from user. Dialog is small popup appears when user need to take a action or show information. Users need to perform an action to close the popup till that it will be visible. PyQt6 provides some common inbuilt dialogs like QMessageBox, QFileDialog, QColorDialog, etc. But user can build their own custom dialog which helps them to achieve lot of things.

Lets see how we can dialog for showing information to user and getting input from user using custom and inbuild dialog.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QHBoxLayout, QMessageBox, QPushButton, QDialog, QLabel, QLineEdit, QComboBox  
  
# Custom Dialog class  
class CustomDialog(QDialog):  
 def \_\_init\_\_(self, parent):  
 super().\_\_init\_\_(parent)  
 self.setWindowTitle('Custom Dialog')  
 self.setMinimumWidth(300)  
 self.setMinimumHeight(300)  
  
 # Layout and widgets  
 layout = QVBoxLayout()  
 label1 = QLabel("Enter your name:", self)  
 nameEdit = QLineEdit(self)  
 label2 = QLabel("Enter your DOB:", self)  
 dobEdit = QLineEdit(self)  
  
 label3 = QLabel("Select Gender:", self)  
 genderCombo = QComboBox(self)  
 genderCombo.addItems(['Male', 'Female'])  
  
 button = QPushButton("OK", self)  
 button.setMaximumWidth(75)  
  
 layout.addWidget(label1)  
 layout.addWidget(nameEdit)  
 layout.addWidget(label2)  
 layout.addWidget(dobEdit)  
 layout.addWidget(label3)  
 layout.addWidget(genderCombo)  
 layout.addWidget(button)  
  
 self.setLayout(layout)  
  
 button.clicked.connect(self.okClicked)  
  
 def okClicked(self):  
 # Write logic to process input here  
  
 # Closes the dialog and Accept  
 self.accept()  
  
class DialogWidgetApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 500, 600)  
 self.setWindowTitle('Dialogs Example')  
   
 def initUI(self):  
 # Create a layout  
 hLayout = QHBoxLayout()  
 button1 = QPushButton("Information Dailog")  
 button2 = QPushButton("User Input Dialog")  
 button1.clicked.connect(self.infoDialog)  
 button2.clicked.connect(self.userDialog)  
 hLayout.addWidget(button1)  
 hLayout.addWidget(button2)  
  
 self.setLayout(hLayout)  
  
 def infoDialog(self):  
 # Create QMessageBox to show information   
 dlg = QMessageBox(self)  
 # Set Title for Dialog  
 dlg.setWindowTitle("Information:")  
 # Set Information to display  
 dlg.setText("This is a information Dialog!")  
 # Set Icon  
 dlg.setIcon(QMessageBox.Icon.Information)  
 dlg.exec()  
   
 def userDialog(self):  
 # intialize custom dialog  
 customDialog = CustomDialog(self)  
 # Show custom dialog to get user information  
 customDialog.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = DialogWidgetApp()  
 w.show()  
 app.exec()

**Output:**

A screenshot of a dialog box

Description automatically generated

## Custom widgets and subclassing

PyQt6 allows user to create custom widgets and subclasses to customize the default functionality for complex use cases. Creating a custom widget means creating a new widget class that extends the functionality of an existing widget or introduces new functionality and appearance to create a widget, the process of overriding existing methods of the widget's appearance and its functionality.

Subclass is the process of creating a new class that inherits attributes and behaviors from an existing class to add or modify its features. In PyQt6, you can subclass existing widgets to override their methods and add new properties or behaviors.

Lets see how to create a custom button widget and add a custom functionality to click event.

from PyQt6.QtWidgets import QApplication, QPushButton, QWidget, QVBoxLayout, QLabel  
from PyQt6.QtCore import pyqtSignal, Qt  
  
class CustomButton(QPushButton):  
 # Define Custom signal  
 sendCustomMessage = pyqtSignal(str)  
  
 def \_\_init\_\_(self, title, parent=None):  
 super().\_\_init\_\_(title, parent)  
 self.title = title  
 # Emit custom signal on click  
 self.clicked.connect(self.emitCustomSignal)  
  
 def emitCustomSignal(self):  
 # Emit the custom signal with a message  
 self.sendCustomMessage.emit(self.title + " clicked!")  
  
class AppWindow(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setWindowTitle("Custom Widget")  
 self.setGeometry(200, 200, 400, 300)  
 mainLayout = QVBoxLayout()  
   
 layout1 = QVBoxLayout()  
 button1 = CustomButton("Layout 1 Button")  
 button1.sendCustomMessage.connect(self.onButtonClick)  
 layout1.addWidget(button1)  
 mainLayout.addLayout(layout1)  
  
 layout2 = QVBoxLayout()  
 button2 = CustomButton("Layout 2 Button")  
 button2.sendCustomMessage.connect(self.onButtonClick)   
 layout2.addWidget(button2)  
 mainLayout.addLayout(layout2)  
   
 layout3 = QVBoxLayout()  
 self.label = QLabel("Label")  
 self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)  
 layout3.addWidget(self.label)  
 mainLayout.addLayout(layout3)  
 self.setLayout(mainLayout)  
   
 def onButtonClick(self, message):  
 self.label.setText(message)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 app = QApplication([])  
 window = AppWindow()  
 window.show()  
 app.exec()

Output:

A screenshot of a computer

Description automatically generated

Styling and Theming in PyQt6

PyQt6 allows you to style and apply theme to your application which will help you to make visually appealing. You can style the application using CSS like web development. You can apply the style in application globally or for induvial widgets. When doing styling its better to reuse the style which will reduce the development time and easy to maintain the application styles. Once you done with styling you need to test application styles across different platforms to ensure a consistent user experience.

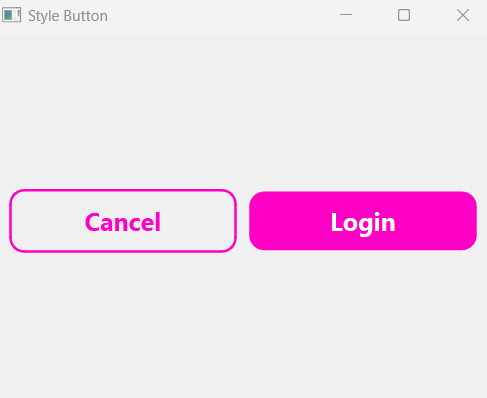
## Using style sheets to customize widget appearance

You can apply CSS to individual widgets to make it look unique. In case we apply css to widget then the similar widget type styles wont change. Basically applying css globally is easy but when we need to override a certain style we can apply the styles individually.

Lets write a program and css how we can styles to modify the button.

from PyQt6.QtWidgets import QApplication, QPushButton, QWidget, QVBoxLayout, QHBoxLayout, QLabel  
  
class StyleButton(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.setWindowTitle("Style Button")  
 self.setGeometry(200, 200, 400, 300)  
 mainLayout = QVBoxLayout()  
   
 layout = QHBoxLayout()  
 button1 = QPushButton("Cancel")  
 # Set Style For Button  
 button1.setStyleSheet(  
 "border: 2px solid #ff00c4;"  
 "color: #ff00c4;"  
 "font-weight: bold;"  
 "padding: 10px;"  
 "text-align: center;"  
 "font-size: 20px;"  
 "margin: 4px 2px;"  
 "border-radius: 12px;"  
 )  
  
 layout.addWidget(button1)  
 mainLayout.addLayout(layout)  
  
 button2 = QPushButton("Login")  
 # Set Style For Button  
 button2.setStyleSheet(  
 "background-color: #ff00c4;"  
 "color: white;"  
 "font-weight: bold;"  
 "padding: 10px;"  
 "text-align: center;"  
 "font-size: 20px;"  
 "margin: 4px 2px;"  
 "border-radius: 12px;"  
 )  
 layout.addWidget(button2)  
  
 self.setLayout(mainLayout)  
   
if \_\_name\_\_ == "\_\_main\_\_":  
 app = QApplication([])  
 window = StyleButton()  
 window.show()  
 app.exec()

Output:



## Applying themes to your PyQt6 application

Applying styles for each item in application is a time-consuming work and increase development time. Its good to write global CSS for the application and write separate styles for widgets where it needs to look different compared to the one applied in common. When we have global CSS it will be easy to do the theming where user can select different themes for the application but they don’t need to rewrite all the CSS again for each theme. You can switch between themes dynamically by applying different style sheets.

PyQt6 Signals and Slots

### Understanding signals and slots in PyQt6

Signals are used for communication between widgets. when there is user interaction like mouse clicked, key pressed, or certain conditions met the signals are emitted. When emitting signals, you can send data too. Signals can be connected to one or more slot. Slot is nothing but a function connected to signal which is called when signal emitted. Qt widgets has its own inbuilt slots you can use. User can connect more than one slots to a signal, each slot will be called one after the other in order. User can connect a slot to receive the signal and disconnect the slot when they no longer need to receive the signal.

### Connecting signals to slots

When user clicks a on a button Clicked signal get emitted, to handle the click signal we must connect a slot function so that the function gets called.

Lets write a program and see how we can connect a slot to the signal.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton  
  
class SignalSlotApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
   
 def initUI(self):  
 # Create a layout  
 layout = QVBoxLayout()  
   
 # Create a button  
 self.button = QPushButton('Click', self)  
  
 layout.addWidget(self.button)  
   
 # Add a Slot to the Signal   
 self.button.clicked.connect(self.onButtonClick)  
 self.setLayout(layout)  
  
 # Slot function  
 def onButtonClick(self):  
 print('Button clicked!')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = SignalSlotApp()  
 w.show()  
 app.exec()

### Emitting custom signals

In an application user might want to create their own signal to handle some custom functionality for that they can use **pyqtSignal**.

**customSignal = pyqtSignal(str)**

**customSignal.emit("custom signal trigered")**

Lets write a program and see how we can trigger a custom signal, connect and disconnect to the signal using slot.

from PyQt6.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QLabel  
from PyQt6.QtCore import Qt, pyqtSignal, QObject, pyqtSignal  
  
class CustomSignal(QObject):  
 # Custom Signal  
 signal = pyqtSignal(str)  
  
class ChildWidget(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.customSignal = CustomSignal()  
   
 layout = QVBoxLayout()  
  
 # Create a button  
 button = QPushButton("Click Me", self)  
 button.clicked.connect(self.onButtonClick)  
 self.count = 0  
  
 layout.addWidget(button)  
 self.setLayout(layout)  
  
 def onButtonClick(self):  
 self.count = self.count + 1  
 # Emit the custom signal with a value  
 self.customSignal.signal.emit(str(self.count))  
  
  
class CustomSignalSlotApp(QWidget):  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.initUI()  
 self.setGeometry(100, 100, 400, 300)  
   
 def initUI(self):  
 self.childWidget = ChildWidget()  
  
 layout = QVBoxLayout()  
 layout.addWidget(self.childWidget)  
 self.parentLabel = QLabel("Click Count 0")  
 layout.addWidget(self.parentLabel, 0, Qt.AlignmentFlag.AlignCenter)  
   
 self.setLayout(layout)  
 # Connect slot to custom signal  
 self.childWidget.customSignal.signal.connect(self.onSignalRecieved)  
   
 def onSignalRecieved(self, value):  
 self.parentLabel.setText("Click Count " + value)  
 if(int(value) == 5):  
 # Disconnect slot from custom signal  
 self.childWidget.customSignal.signal.disconnect(self.onSignalRecieved)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 app = QApplication([])  
 w = CustomSignalSlotApp()  
 w.show()  
 app.exec()

Integrating Scientific Libraries with PyQt6

PyQt6 gives excellent support for integrating scientific libraries, allows user to create powerful and interactive applications to do data analysis, visualization, and scientific computing functionalities. To build scientific applications you can use the support of famous Qt libraries like NumPy, SciPy, Matplotlib, etc, it will help you to perform complex data analysis, mathematical operations, and visualizations.

## Introduction to popular scientific libraries (NumPy, SciPy, Matplotlib, etc.)

## Using scientific libraries in PyQt6 applications

## Visualizing scientific data with PyQt6 and Matplotlib

Signal Processing with PyQt6

## Fundamentals of signal processing

## Introduction to PyQt6's signal processing capabilities

## Processing and filtering signals in real-time applications

Real-Time Data Visualization with PyQt6

## Introduction to real-time data visualization

## Updating PyQt6 widgets dynamically

## Building real-time applications with PyQt6

PyQt6 and Data Analysis

## Loading and processing data in PyQt6 applications

## Analysing data using scientific libraries

## Presenting data analysis results with PyQt6

Advanced PyQt6 Techniques

## Multi-threading in PyQt6 applications

## Internationalization and localization

## Packaging and distributing PyQt6 applications

Case Studies and Practical Examples

## Building a scientific data analysis tool with PyQt6

## Developing a real-time monitoring application

## Showcasing advanced PyQt6 features in real-world scenarios