**Customizing Widgets for User Interaction and Data Handling in PySide6**

PySide6 allows extensive **customization** of widgets to improve **user interactions** and **data handling**. Below, we will explore:

1. **Customizing QTableView** → Editing cells dynamically.
2. **Customizing QTreeView** → Expanding and collapsing nodes.
3. **Customizing QComboBox** → Populating dynamically and handling events.

**1. Customizing QTableView (Editable Table with Custom Data Handling)**

In this example, we make a QTableView editable and store data dynamically.

**Example: Editable QTableView with Custom Model**

from PySide6.QtWidgets import QApplication, QTableView, QVBoxLayout, QWidget, QPushButton

from PySide6.QtGui import QStandardItemModel, QStandardItem

import sys

class EditableTable(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("Custom QTableView Example")

self.resize(400, 300)

layout = QVBoxLayout()

# Create Table Model

self.model = QStandardItemModel(3, 2)

self.model.setHorizontalHeaderLabels(["Name", "Age"])

# Populate table with default data

data = [("Alice", "25"), ("Bob", "30"), ("Charlie", "28")]

for row, (name, age) in enumerate(data):

self.model.setItem(row, 0, QStandardItem(name))

self.model.setItem(row, 1, QStandardItem(age))

# Create Table View

self.table = QTableView()

self.table.setModel(self.model)

layout.addWidget(self.table)

# Button to print table data

self.button = QPushButton("Print Table Data")

self.button.clicked.connect(self.print\_data)

layout.addWidget(self.button)

self.setLayout(layout)

def print\_data(self):

"""Prints the table content"""

for row in range(self.model.rowCount()):

name = self.model.item(row, 0).text()

age = self.model.item(row, 1).text()

print(f"Row {row}: Name={name}, Age={age}")

app = QApplication(sys.argv)

window = EditableTable()

window.show()

sys.exit(app.exec())

**Features & Customizations:**

✅ **Editable cells** using QStandardItemModel.  
✅ **Dynamic data storage** and retrieval.  
✅ **Button to fetch & print table data** for validation.

**2. Customizing QTreeView (Dynamic Node Expansion and Context Menu Handling)**

This example creates a file explorer with **expand/collapse functionality** and a **right-click context menu**.

**Example: Custom QTreeView with Context Menu**

from PySide6.QtWidgets import QApplication, QTreeView, QFileSystemModel, QMenu

from PySide6.QtCore import Qt, QPoint

import sys

class CustomTreeView(QTreeView):

def \_\_init\_\_(self):

super().\_\_init\_\_()

# Set up file system model

self.model = QFileSystemModel()

self.model.setRootPath("")

self.setModel(self.model)

self.setRootIndex(self.model.index("."))

self.setWindowTitle("Custom QTreeView")

self.resize(600, 400)

# Enable right-click context menu

self.setContextMenuPolicy(Qt.CustomContextMenu)

self.customContextMenuRequested.connect(self.show\_context\_menu)

def show\_context\_menu(self, position: QPoint):

"""Creates a right-click context menu."""

menu = QMenu()

expand\_action = menu.addAction("Expand All")

collapse\_action = menu.addAction("Collapse All")

action = menu.exec\_(self.viewport().mapToGlobal(position))

if action == expand\_action:

self.expandAll()

elif action == collapse\_action:

self.collapseAll()

app = QApplication(sys.argv)

tree = CustomTreeView()

tree.show()

sys.exit(app.exec())

**Features & Customizations:**

✅ **Right-click context menu** using QMenu.  
✅ **Expand/Collapse functionality** via context actions.  
✅ **Dynamic filesystem browsing** via QFileSystemModel.

**3. Customizing QComboBox (Dynamic Data Handling & Selection Event)**

This example **populates a QComboBox dynamically** and **reacts to user selection**.

**Example: Dynamic QComboBox with Selection Handling**

from PySide6.QtWidgets import QApplication, QWidget, QVBoxLayout, QComboBox, QLabel, QPushButton

import sys

class DynamicComboBox(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.setWindowTitle("Custom QComboBox Example")

self.resize(300, 200)

layout = QVBoxLayout()

self.label = QLabel("Select an option:")

layout.addWidget(self.label)

self.combo = QComboBox()

self.combo.addItems(["Python", "C++", "Java", "JavaScript"])

self.combo.currentIndexChanged.connect(self.on\_selection\_change)

layout.addWidget(self.combo)

self.button = QPushButton("Add Item")

self.button.clicked.connect(self.add\_new\_item)

layout.addWidget(self.button)

self.setLayout(layout)

def on\_selection\_change(self, index):

"""Handles dropdown selection changes."""

self.label.setText(f"Selected: {self.combo.itemText(index)}")

def add\_new\_item(self):

"""Adds a new item to the ComboBox dynamically."""

new\_item = f"NewLang {self.combo.count() + 1}"

self.combo.addItem(new\_item)

app = QApplication(sys.argv)

window = DynamicComboBox()

window.show()

sys.exit(app.exec())

**Features & Customizations:**

✅ **Handles selection changes dynamically.**  
✅ **Allows adding new items at runtime.**  
✅ **Updates QLabel with selected option.**

**Conclusion:**

These examples demonstrate how **PySide6 widgets can be customized** for better user interaction and data handling:

* **QTableView**: Editable tables with dynamic data handling.
* **QTreeView**: File explorer with expandable/collapsible nodes & a context menu.
* **QComboBox**: Dynamically adding new items and handling selections.