

Ventanas, botones, etiquetas, cuadros de texto.


Recapitulemos los Widgets más importantes:

Widget	Qué hace
QLabel	Solo una etiqueta, no interactiva
QLineEdit	Cuadro para introducir una línea de texto
QCheckBox	Una casilla de verificación
QRadioButton	Un grupo con una sola opción activa
QPushButton	Un botón
QComboBox	Un cuadro de lista desplegable
QLCDNumber	Una pantalla LCD
QDateEdit	Para editar fechas
QDateTimeEdit	Para editar fechas y fechas y horas
QDial	Esfera giratoria
QSpinBox	Un girador de números enteros
QDoubleSpinBox	Un girador de números decimales
QFontComboBox	Una lista de fuentes
QProgressBar	Una barra de progreso
QSlider	Control deslizante
QTimeEdit	Para editar tiempos

Lista completa:

Qt Widgets C++ Classes | Qt Widgets 5.15.17

The Qt Widgets module extends Qt GUI with C++ widget functionality. More...

 <https://doc.qt.io/qt-5/qtwidgets-module.html>

▼ QLabel

(Archivo e01)

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Mi Aplicación")
        widget = QLabel("Hola")
        # Tomamos el objeto "fuente" del widget para poder modificarlo y
        devolverlo al widget
        font = widget.font()
        font.setPointSize(30)
        widget.setFont(font)
        # Alineación horizontal y vertical del texto
        widget.setAlignment(
            Qt.AlignmentFlag.AlignHCenter | Qt.AlignmentFlag.AlignVCente
```

```
r)

self.setCentralWidget(widget)
```

También se pueden añadir imágenes desde un archivo de esta manera:

```
widget.setPixmap(QPixmap("otje.jpg"))
# Para que la imagen se estire y encoja con la ventana
widget.setScaledContents(True)
```

▼ QLineEdit

QLineEdit es un widget de PyQt que proporciona un campo de texto de una sola línea para capturar contenido escrito por el usuario. Es un componente común en aplicaciones gráficas que requieren la entrada de texto, como formularios de registro, búsqueda, edición de texto, etc.

Características

- Permite capturar texto de una sola línea.
- Admite deshacer y rehacer, cortar y pegar, y arrastrar y soltar.
- Tiene un marco predeterminado que se puede configurar mediante el método `setFrame`.
- Ofrece varias señales que permiten responder a eventos como cambios en el texto, presiones de teclas (como Enter o Backspace), y selección de texto.
- Puede ser personalizado con atributos como `maxLength` (longitud máxima del texto), `placeholderText` (texto de ayuda o placeholder), y `echoMode` (modo de visualización del texto, como texto visible o texto oculto).

Uso

Para utilizar QLineEdit en una aplicación PyQt, debes crear un objeto de la clase `QLineEdit` y agregarlo a un layout o contenedor de widgets. Luego, puedes configurar sus atributos y conectar señales para responder a eventos.

Ejemplo

A continuación, se muestra un ejemplo básico de cómo crear un QLineEdit y configurarlo:

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QLineEdit

class VentanaPrincipal(QMainWindow):
    def __init__(self):
        super().__init__()
        self.texto = QLineEdit()
        self.texto.setPlaceholderText("Escribe algo")
        self.texto.setMaxLength(20)
        self.setCentralWidget(self.texto)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    ventana = VentanaPrincipal()
```

```
ventana.show()
sys.exit(app.exec_())
```

En este ejemplo, se crea un objeto `QLineEdit` y se configura con un texto de ayuda (`placeholderText`) y una longitud máxima (`maxLength`). Luego, se agrega al centro de la ventana principal (`setCentralWidget`).

Conectar señales

Para responder a eventos como cambios en el texto o presiones de teclas, puedes conectar señales a métodos de instancia. Por ejemplo:

```
self.texto.textChanged.connect(self.texto_cambiado)
self.texto.returnPressed.connect(self.enter_presionado)
```

Donde `texto_cambiado` y `enter_presionado` son métodos que se ejecutarán cuando se produzcan los eventos correspondientes.

En resumen, `QLineEdit` es un widget fundamental en PyQt para la entrada de texto y ofrece varias características y opciones para personalizar su comportamiento y apariencia.

▼ QCheckBox

(Archivo e02)

Es una casilla que el usuario puede marcar. Sin embargo, como ocurre con todos los widgets de Qt, existen varias opciones configurables para cambiar su comportamiento.

```
class VentanaPrincipal(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Mi Aplicación")
        widget = QCheckBox("Esto es un checkbox")
        # widget.setCheckState(Qt.CheckState.Checked)
        # Existe un tercer estado intermedio
        # Así: widget.setCheckState(Qt.PartiallyChecked)
        # O así:
        widget.setTristate(True)
        widget.stateChanged.connect(self.show_state)
        self.setCentralWidget(widget)

    def show_state(self, estado):
        print(Qt.CheckState(estado) == Qt.CheckState.Checked)
        print(f"El estado es {estado}")
```

▼ QRadioButton

`QRadioButton` es un widget de PyQt que presenta una opción de selección múltiple. Estos botones se utilizan comúnmente para presentar al usuario una elección entre varias opciones.

En un grupo de botones de radio, solo uno puede estar seleccionado a la vez; si el usuario selecciona otro botón, el anteriormente seleccionado se deselecciona.

Características

- Los botones de radio son autoexclusivos por defecto. Esto significa que si se crean botones de radio en el mismo contenedor padre, solo uno puede estar seleccionado a la vez.
- Los botones de radio emiten el señal `toggled()` cada vez que se cambia su estado (seleccionado/deseleccionado).
- Se pueden establecer texto y iconos en los botones de radio utilizando los métodos `setText()` y `setIcon()`.
- Se pueden establecer atajos de teclado (shortcut keys) precediendo el carácter preferido con un ampersand (&).

Uso

Para utilizar un botón de radio en PyQt, se crea un objeto `QRadioButton` y se agrega a un layout (como `QVBoxLayout` o `QHBoxLayout`). Se puede establecer el texto y el estado inicial del botón utilizando los métodos `setText()` y `setChecked()`.

Ejemplo

Aquí tienes un ejemplo simple de cómo utilizar un botón de radio en PyQt:

```
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QRadioButton

class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        layout = QVBoxLayout()
        self.setLayout(layout)

        rb1 = QRadioButton("Opción 1", self)
        rb2 = QRadioButton("Opción 2", self)
        rb3 = QRadioButton("Opción 3", self)

        layout.addWidget(rb1)
        layout.addWidget(rb2)
        layout.addWidget(rb3)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec_())
```

En este ejemplo, se crean tres botones de radio y se agregan a un layout vertical. Cada botón tiene un texto diferente y se puede seleccionar uno a la vez.

Estilos y personalización

Los botones de radio también admiten estilos y personalización utilizando CSS-like syntax (QSS) o estilos en línea. Por ejemplo, se puede cambiar el color del fondo, el tamaño del texto y el diseño del botón.

▼ QPushButton (revisar)

En PyQt, `QPushButton` es un widget de botón que permite a los usuarios interactuar con su aplicación mediante clicks. A continuación, se presentan los conceptos clave y métodos relacionados con `QPushButton`:

Propiedades

- `text`: El texto que se muestra en el botón.
- `icon`: El icono que se muestra junto al texto del botón.
- `shortcut`: La tecla o combinación de teclas asociada al botón.
- `default`: Indica si el botón es el botón predeterminado en la ventana contenedora.
- `flat`: Indica si el borde del botón está elevado o no.

Métodos

- `setText(text)`: Establece el texto del botón.
- `text()`: Devuelve el texto actual del botón.
- `setIcon(icon)`: Establece el icono del botón.
- `icon()`: Devuelve el icono actual del botón.
- `setShortcut(shortcut)`: Establece la tecla o combinación de teclas asociada al botón.
- `shortcut()`: Devuelve la tecla o combinación de teclas actualmente asociada al botón.
- `setEnabled(enabled)`: Habilita o deshabilita el botón.
- `isEnabled()`: Devuelve el estado de habilitación actual del botón.

Señales y Slots

`QPushButton` emite señales cuando se produce un evento, como un click o un cambio en el estado de habilitación. Los slots pueden ser conectados a estas señales para realizar acciones específicas cuando se produce el evento.

Ejemplo

Supongamos que queremos crear un botón con el texto "Limpiar" que, cuando se hace clic, borra el contenido de un `QLineEdit`. Podríamos hacerlo de la siguiente manera:

```
import sys
from PyQt5.QtWidgets import QApplication, QLineEdit, QPushButton, QMainWindow, QWidget

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.ui = VentanaPrincipal()
        self.ui.setupUi(self)
        self.setWindowFlags(Qt.WindowMinimizeButtonHint | Qt.WindowClose
```

```

ButtonHint | Qt.MSWindowsFixedSizeDialogHint)
        self.setFixedSize(1050, 750)

        self.ui.mni_netejar = QPushButton("Limpiar")
        self.ui.mni_netejar.clicked.connect(self.limpiar)
        self.ui.linEd_QIT_03A.setText("123456")

    def limpiar(self):
        self.ui.linEd_QIT_03A.setText("")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    GUI = MainWindow()
    GUI.show()
    sys.exit(app.exec_())

```

En este ejemplo, creamos un botón con el texto "Limpiar" y lo conectamos a un slot `limpiar` que borra el contenido del `QLineEdit` cuando se hace clic en el botón.

Módulos Externos

Si deseas crear un módulo externo que contenga una función para limpiar el contenido de un `QLineEdit`, puedes hacerlo de la siguiente manera:

```

# limpiar.py
def limpiar(parent):
    parent.linEd_QIT_03A.setText("")

```

Luego, en tu aplicación principal, puedes importar este módulo y utilizar la función `limpiar` como slot para el botón:

```

import sys
from PyQt5.QtWidgets import QApplication, QLineEdit, QPushButton, QMainWindow
from limpiar import limpiar

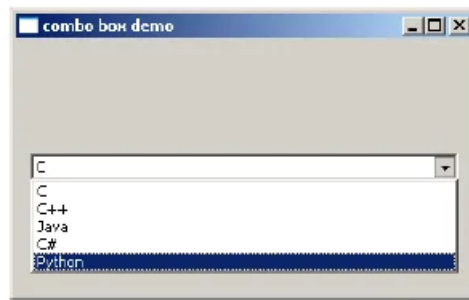
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        ...
        self.ui.mni_netejar.clicked.connect(limpiar)
        ...

```

▼ QComboBox

<https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QComboBox.html>

<https://doc.qt.io/qt-6/qcombobox.html>



El widget `QComboBox` en PyQt es una lista desplegable de elementos que permite a los usuarios seleccionar una opción de un conjunto de valores predefinidos. Es un widget versátil y ampliamente utilizado en aplicaciones GUI.

Características principales

1. **Añadir Items:** los elementos se pueden agregar individualmente usando `addItem()` o varios de golpe usando el método `addItems()`.
2. **Edición:** `QComboBox` se puede hacer editable estableciendo la propiedad `editable` en `True`. Los usuarios pueden ingresar sus propios valores personalizados.
3. **Política de inserción:** la propiedad `insertPolicy` determina cómo se insertan los nuevos valores en la lista. Las opciones incluyen:
 - `QComboBox.NoInsert`: los nuevos valores no se insertan en la lista.
 - `QComboBox.InsertAtTop`: los nuevos valores se insertan en la parte superior de la lista.
 - `QComboBox.InsertAtBottom`: los nuevos valores se insertan en la parte inferior de la lista.
 - `QComboBox.InsertAlphabetically`: los nuevos valores se insertan en orden alfabético.
4. **Signals:** Entre las señales que puede emitir `QComboBox` están:
 - `currentIndexChanged`: se emite cuando cambia el elemento seleccionado actualmente.
 - `currentTextChanged`: se emite cuando cambia el texto del elemento seleccionado actualmente.
 - `activated`: se emite cuando se selecciona o resalta un elemento.
5. **Obtención del estado actual:** puede recuperar el estado actual de `QComboBox` utilizando métodos como:
 - `currentIndex()`: devuelve el índice del elemento seleccionado actualmente.
 - `currentText()`: devuelve el texto del elemento seleccionado actualmente.
 - `itemText()`: devuelve el texto de un elemento en un índice especificado.
 - `count()`: devuelve la cantidad de elementos en la lista.

Código de ejemplo

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QComboBox

class VentanaPrincipal(QMainWindow):
    def __init__(self):
```

```

        super().__init__()

        self.combo_box = QComboBox(self)
        self.combo_box.addItem("Option 1")
        self.combo_box.addItem("Option 2")
        self.combo_box.addItem("Option 3")
        self.combo_box.currentIndexChanged.connect(self.selection_changed)

    def selection_changed(self, index):
        print(f"Selected item: {self.combo_box.currentText()}")

app = QApplication(sys.argv)
ventana = VentanaPrincipal()
ventana.show()
sys.exit(app.exec_())

```

Este código crea un QComboBox con tres opciones y conecta su señal `currentIndexChanged` a un método `selection_changed`, que imprime el texto del elemento seleccionado actualmente en la consola.

Puedo crear el ComboBox en otro archivo/módulo de python e importarlo desde el archivo principal en el que tengo la ventana principal.

```

#Crear un Combo box
my_combo = QtWidgets.QComboBox(self) #Añadir elementos
my_combo.addItem("Pepperoni", "Something")
my_combo.addItem("Cheese", 2)
my_combo.addItem("Mushroom", QtWidgets.QWidget)
my_combo.addItem("Peppers")
my_combo.addItem("w", "x", "y")
# Podemos añadir los items en la posición que queramos
my_combo.insertItem(3, "z")
my_combo.insertItems(3, ["a", "b"])

#Colocar combobox en la ventana
self.layout().addWidget(my_combo)

```

▼ QLCDNumber

QLCDNumber es un widget de PyQt que muestra un número con dígitos similares a los de una pantalla LCD. Permite mostrar números en diferentes formatos, como decimal, hexadecimal, octal y binario.

Estilos

El widget QLCDNumber admite tres estilos para mostrar los dígitos:

- **Outline:** produce segmentos elevados llenos de color de fondo.
- **Filled** (por defecto): produce segmentos elevados llenos de color del texto.
- **Flat:** produce segmentos planos llenos de color del texto.

Métodos

QLCDNumber proporciona varios métodos para interactuar con el widget:

- `display(QString constante)` : muestra un texto como una cadena de caracteres.
- `display(int num)` : muestra un número entero.
- `display(double num)` : muestra un número decimal.
- `value() const` : devuelve el valor actual del widget.

Uso

Para utilizar QLCDNumber en un proyecto PyQt, debes importar el módulo `QtGui` y crear un objeto `QLCDNumber`. Luego, puedes configurar el estilo y el valor inicial del widget utilizando los métodos mencionados anteriormente. Finalmente, puedes conectar el método `display()` a un evento, como un timer, para actualizar el valor del widget en tiempo real.

Ejemplo

Aquí tienes un ejemplo básico de cómo utilizar QLCDNumber en PyQt:

```
import sys
from PyQt5 import QtGui, QtCore

class MainWindow(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_ui()

    def init_ui(self):
        self.setWindowTitle("PyQt5 QLCD Number")
        self.setGeometry(200, 500, 400, 300)

        self.lcd = QtGui.QLCDNumber(self)
        self.lcd.setDigitCount(8) # número de dígitos a mostrar
        self.lcd.display(0.0) # valor inicial

        self.timer = QtCore.QTimer(self)
        self.timer.timeout.connect(self.update_lcd)
        self.timer.start(1000) # actualizar cada 1 segundo

    def update_lcd(self):
        # obtener valor actualizado desde una fuente (por ejemplo, un archivo Excel)
        value = 12.34 # ejemplo de valor actualizado
        self.lcd.display(value)

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = MainWindow()
```

```
window.show()  
sys.exit(app.exec_())
```

Este ejemplo crea un ventana con un widget QLCDNumber y configura un timer para actualizar el valor del widget cada segundo. En este caso, se muestra un valor decimal, pero puedes cambiar el formato y estilo según necesites.

