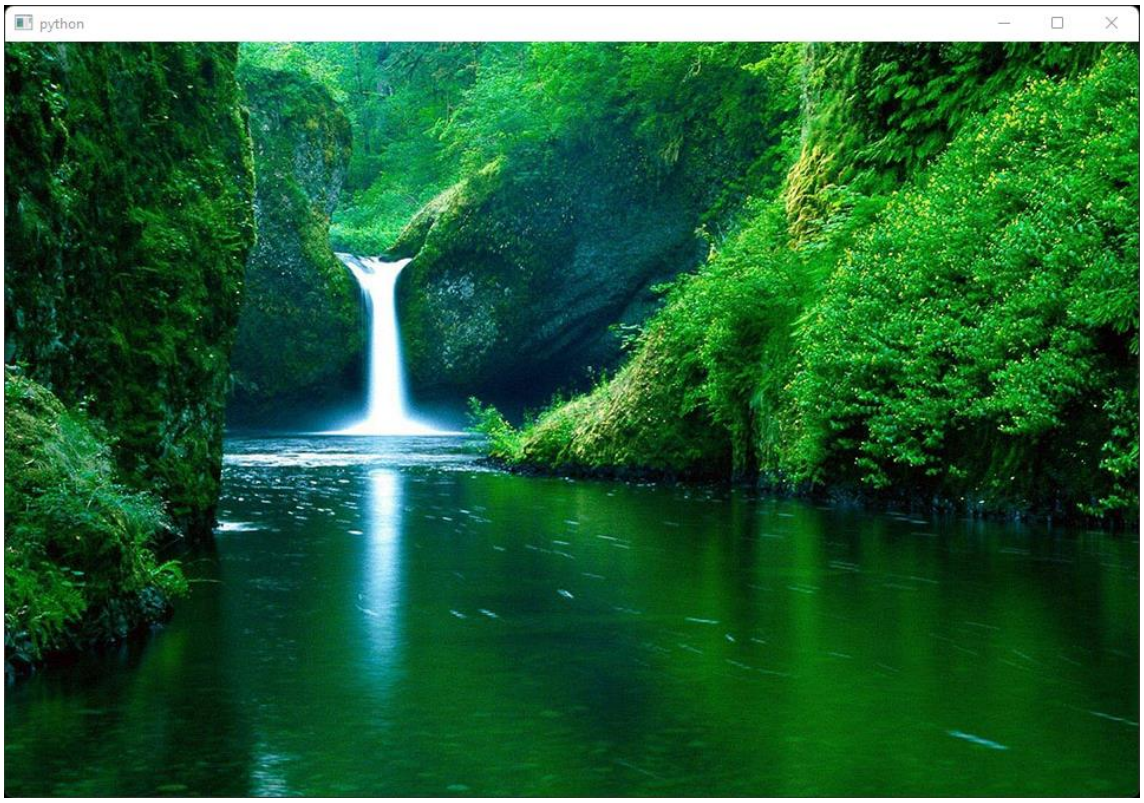
Etiquetas QLabel



Empezamos este tour con las etiquetas, uno de los widgets más sencillos de Qt. Se trata de una pieza de texto que se puede posicionar en nuestra aplicación. Podemos asignar el texto al crearlas o mediante su método `setText()`:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QLabel
from PySide6.QtCore import QSize
import sys


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setMinimumSize(QSize(480, 320))

        # widget etiqueta
        etiqueta = QLabel("Soy una etiqueta")
        # establecemos el widget central
        self.setCentralWidget(etiqueta)


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

Las etiquetas permiten configurar una fuente a través de la cuál controlar el tamaño y otros atributos.

Podemos recuperar la fuente por defecto y aumentar su tamaño:

```
# recuperamos la fuente por defecto
fuente = etiqueta.font()
# establecemos un tamaño
fuente.setPointSize(24)
# la asignamos a la etiqueta
etiqueta.setFont(fuente)
```

O podemos utilizar una fuente del sistema, aunque para ello debemos crear una instancia de la clase `QFont`:

```
from PySide6.QtGui import QFont  # nuevo

# cargamos una fuente del sistema
fuente = QFont("Comic Sans MS", 24)
# la asignamos a la etiqueta
etiqueta.setFont(fuente)
```

Las etiquetas también nos permiten alinearlas respecto a su contenedor, para ello necesitamos importar las definiciones estándard de Qt:

```
from PySide6.QtCore import QSize, Qt  # editado

# establecemos unas flags de alineamiento
etiqueta.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
```

Estas definiciones se conocen como banderas o "flags". Son enumeradores con nombre para usarlos más cómodamente. Al unirlos con la tubería se evalúan en conjunto:

```
print(int(Qt.AlignHCenter), int(Qt.AlignVCenter), int(Qt.AlignHCenter
| Qt.AlignVCenter))
print(bin(Qt.AlignHCenter), bin(Qt.AlignVCenter), bin(Qt.AlignHCenter
| Qt.AlignVCenter))
```

Otra cosa muy útil que permiten las etiquetas es cargar imágenes en su interior, para ello utilizamos un objeto de tipo `QPixemap` o mapa de píxeles creado a partir de una imagen y lo asignamos a la etiqueta. Tengo una imagen preparada en el directorio del script:

```
from PySide6.QtGui import QFont, QPixmap  # editado

# creamos la imagen
imagen = QPixmap("naturaleza.jpg")
# la asginamos a la etiqueta
etiqueta.setPixmap(imagen)
```

En el momento en que cargamos recursos externos debemos empezar a tener en cuenta el concepto de la ruta al recurso.

Cuando establecemos `naturaleza.jpg`, el programa espera que ese recurso se encuentre en el mismo directorio desde donde se ejecuta el script. Por eso debemos tener en cuenta es posible ejecutar un script sin estar en su mismo directorio y si lo hacemos esos recursos no se encontrarán. Debemos navegar hasta el programa y ejecutarlo desde su carpeta:

```
cd 1-2/1-2-1
python programa.py
```

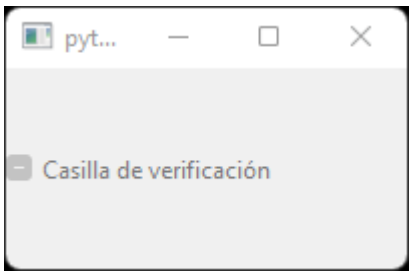
Alternativamente podemos utilizar el módulo `Path` de Python para generar una ruta absoluta al recurso concreto a partir del script actual y solventar el problema para siempre. Os recomiendo crear una función como la siguiente para facilitar la reutilización:

```
def absPath(file):
    # Devuelve la ruta absoluta a un fichero desde el propio script
    return str(Path(__file__).parent.absolute() / file)

print(absPath("naturaleza.jpg"))
imagen = QPixmap(absPath("naturaleza.jpg"))
```

Sea como sea nuestra imagen ya se muestra, pero si quisiéramos que se reescale junto al tamaño de la ventana deberíamos establecer el atributo `scaledContents` en `True`:

```
# hacemos que se escale con la ventana
etiqueta.setScaledContents(True)
```

Casillas QCheckBox



Seguimos el tour viendo las casillas de verificación o checkboxes:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QCheckBox  # edited
from PySide6.QtCore import QSize, Qt
import sys


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

        # creamos una casilla y la establecemos de widget central
        casilla = QCheckBox("Casilla de verificación")
        self.setCentralWidget(casilla)


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

Sirven como alternadores para saber si una opción está marcada o no. Podemos conectar una señal `stateChanged` para saber cuando cambia y consultar su valor:

```
# señal para detectar cambios en la casilla
casilla.stateChanged.connect(self.estado_cambiado)
```

```
def estado_cambiado(self, estado):
    print(estado)
```

Fijaos que curiosamente los estados numéricos de la casilla son 0 y 2. Cuando es cero la está desmarcada y cuando es dos está marcada. Podemos utilizar banderas para analizar de forma más amigable la casilla:

```
def estado_cambiado(self, estado):
    if estado == Qt.Checked:
        print("Casilla marcada")
    if estado == Qt.Unchecked:
        print("Casilla desmarcada")
```

Y por curiosidad, ¿si desmarcado es 0 y marcado 2, a qué hace referencia el estado 1? Este estado se llama tri-estado e indica que una casilla no está estrictamente ni marcada ni desmarcada, sino en estado neutro:

```
# establecemos el triestado por defecto, también funcionan los otros
casilla.setCheckState(Qt.PartiallyChecked)

def estado_cambiado(self, estado):
    if estado == Qt.Checked:
        print("Casilla marcada")
    if estado == Qt.Unchecked:
        print("Casilla desmarcada")
    if estado == Qt.PartiallyChecked:
        print("Casilla neutra")
```

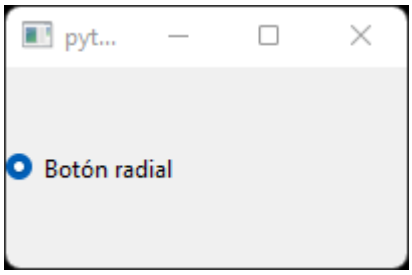
También podemos desactivar la casilla utilizando su método `setEnabled`, que es común en la mayoría de widgets:

```
# la podemos desactivar
casilla.setEnabled(False)
```

Y para consultar el estado actual de la casilla simplemente utilizaríamos `isChecked` o `isTristate` para especificar un estado neutro:

```
# consultamos el valor actual
print("¿Activada?", casilla.isChecked())
print("¿Neutra?", casilla.isTristate())
```

Botones radiales QRadioButton

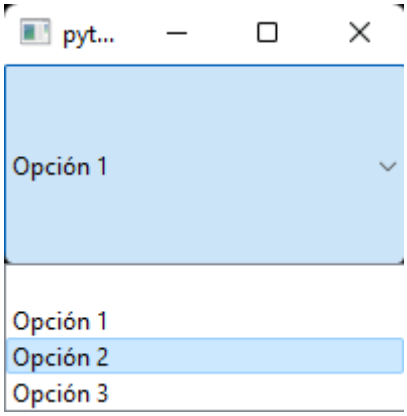


Muy parecidas a las casillas son los botones radiales, sin embargo estos solo pueden marcarse o desmarcarse, no tienen estado neutro:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QRadioButton
from PySide6.QtCore import QSize, Qt
import sys
```

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

        # creamos un botón radial y lo establecemos de widget central
        radial = QRadioButton("Botón radial")
        self.setCentralWidget(radial)

        # señal para detectar cambios en el botón
        radial.toggled.connect(self.estado_cambiado)

        # Podemos activarla por defecto
        radial.setChecked(True)

        # consultamos el valor actual
        print("¿Activada?", radial.isChecked())

    def estado_cambiado(self, estado):
        if estado:
            print("Radial marcado")
        else:
            print("Radial desmarcado")
```

Desplegables QComboBox



Los desplegables son listas de opciones de las cuales se pueden selecciona una única opción.

Para añadir opciones se utiliza su método `addItems`:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QComboBox  # edited
from PySide6.QtCore import QSize, Qt
import sys


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

        # creamos un desplegable
        desplegable = QComboBox()
```

```
        self.setCentralWidget(desplegable)

        desplegable.addItems(["Opción 1", "Opción 2", "Opción 3"])
```

Dependiendo de si queremos consultar el índice o el valor al cambiar podemos usar una señal `currentTextChanged` o `currentIndexChanged`:

```
desplegable.currentIndexChanged.connect(self.indice_cambiado)
desplegable.currentTextChanged.connect(self.texto_cambiado)

def indice_cambiado(self, indice):
    print("Nuevo índice ->", indice)

def texto_cambiado(self, texto):
    print("Nuevo texto ->", texto)
```

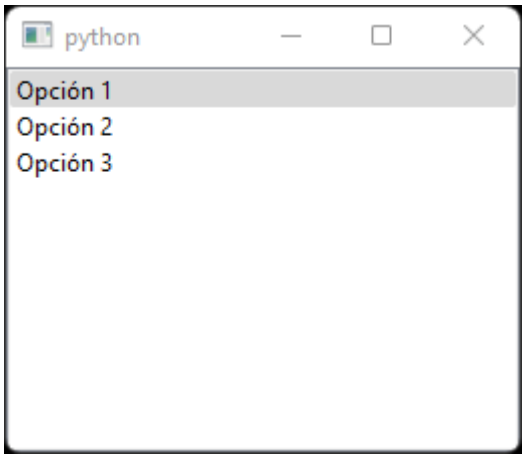
Si quisiéramos un valor vacío por defecto, ya que éste se establece como el primer elemento añadido, sería tan sencillo como poner una cadena vacía:

```
desplegable.addItems(["", "Opción 1", "Opción 2", "Opción 3"])
```

Y para comprobar la opción seleccionada:

```
# consultamos el valor actual
print("Índice actual ->", desplegable.currentIndex())
print("Texto actual ->", desplegable.currentText())
```

Listas QListWidget



Las listas son muy parecidas a los desplegables pero aquí las opciones no están ocultas ni hay ninguna activa por defecto. En lugar de índices manejan un tipo de valor llamado `QItem` y la señal de cambio aquí es `currentItemChanged`:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QListWidget
from PySide6.QtCore import QSize, Qt
import sys


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

        # creamos una lista
```

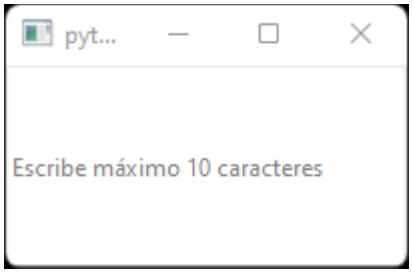
```
        lista = QListWidget()
        self.setCentralWidget(lista)

        # Añadimos algunas opciones
        lista.addItems(["Opción 1", "Opción 2", "Opción 3"])

        # Y algunas señales
        lista.currentItemChanged.connect(self.item_cambiado)

    def item_cambiado(self, item):
        # Conseguimos el texto del ítem con su método text()
        print("Nuevo ítem ->", item.text())
```

Y para conseguir el ítem actual utilizaremos:

```
print(lista.currentItem())
```

Como por defecto la lista no tiene nada seleccionado muestra `None`.

Campos de texto QLineEdit



Los campos de texto son los widgets que permiten capturar contenido escrito por el usuario:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QLineEdit
from PySide6.QtCore import QSize, Qt
import sys


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

        # creamos un campo de texto
        texto = QLineEdit()
        self.setCentralWidget(texto)

        # Probamos algunas opciones
        texto.setMaxLength(10)
        texto.setPlaceholderText("Escribe máximo 10 caracteres")
```

Podemos probar algunas señales para practicar:

```
# Probamos algunas señales
texto.textChanged.connect(self.texto_cambiado)
texto.returnPressed.connect(self.enter_presionado)

def texto_cambiado(self, texto):
```

```
    print("Texto cambiado ->", texto)

def enter_presionado(self):
    # al presionar enter recuperamos el texto a partir del widget
central
    texto = self.centralWidget().text()
    print("Enter presionado, texto ->", texto)
```

Campos numéricos QSpinBox y QDoubleSpinBox

A diferencia de los campos de texto, los numéricos fuerzan al usuario a escribir números y proveen métodos para su control. Encontramos para almacenar enteros y decimales, empecemos por los enteros:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QSpinBox
from PySide6.QtCore import QSize, Qt
import sys


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

        # creamos un campo numérico entero
        numero = QSpinBox()
        self.setCentralWidget(numero)

        # Probamos algunas opciones
        numero.setMinimum(0)
        numero.setMaximum(10)
        numero.setRange(0, 10)
        numero.setSingleStep(1)

        # Probamos algunas señales
        numero.valueChanged.connect(self.valor_cambiado)

    def valor_cambiado(self, numero):
        # al presionar enter recuperamos el texto a partir del widget
central
        print("Valor cambiado ->", numero)
```

Los campos numéricos permiten establecer prefijos y sufijos, útiles para manejar monedas y medidas:

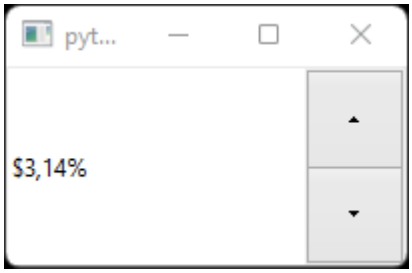
```
numero.setPrefix("$")
numero.setSuffix("%")
```

En cuanto a los decimales son exactamente lo mismo pero utilizando el widget `QDoubleSpinBox`:

- `QSpinBox` -> `QDoubleSpinBox`
- `numero = QSpinBox()` -> `numero = QDoubleSpinBox()`
- `numero.setSingleStep(1)` -> `numero.setSingleStep(0.5)`

En ambos widgets podemos establecer un valor por defecto con el método `setValue` y recuperarlo con `value`:

```
# Establecer y recuperar el valor
numero.setValue(3.14)
print(numero.value())
```



Con esto acabamos el repaso de los widgets básicos para formularios.

Existen muchos otros widgets para realizar todo tipo de tareas. Descubriremos algunos de ellos más adelante, pero si queréis una referencia completa lo mejor es buscar en la documentación de qt las clases heredadas de `QWidget`.