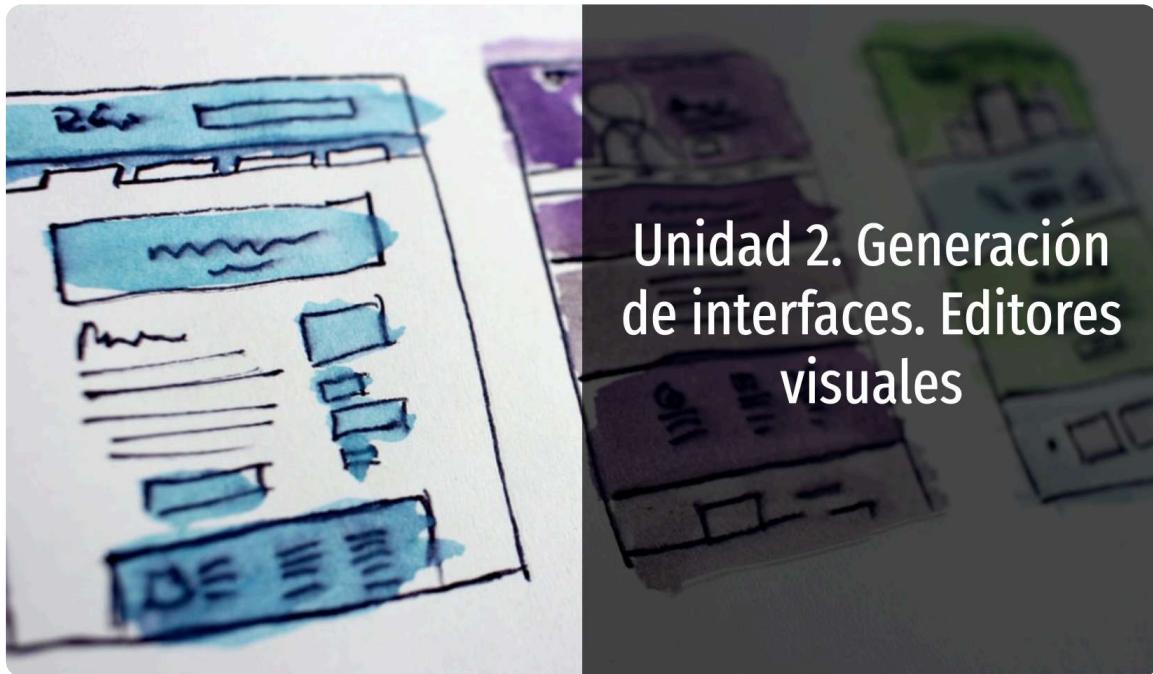


## TEMA 2 - GENERACIÓN DE INTERFACES. EDITORES VISUALES.



## TEMA 2 - GENERACIÓN DE INTERFACES. EDITORES VISUALES.

### Índice de contenidos

1. Introducción.
2. Componentes de uso común.
3. Contenedores de componentes. Diseño.
4. Barras de herramientas, barra de estado y menús.
5. Diálogos y otras ventanas.

### Objetivos de aprendizaje

1. Crear interfaces gráficas.
2. Ubicar los componentes y modificar sus propiedades.
3. Analizar y modificar código ya generado.
4. Asociar acciones a los eventos de los componentes de la interfaz.
5. Desarrollar aplicaciones con interfaz gráfica.

# 1. Introducción

Cada día aparecen nuevas aplicaciones que facilitan la vida a las personas o ayudan a resolver algún problema. Actualmente existen, y siguen apareciendo día a día, tecnologías con las que desarrollar estas aplicaciones.

En esta unidad haremos uso de Python y del *framework* Qt para desarrollar nuestras propias aplicaciones con interfaz gráfica de usuario.

En este primer apartado de la unidad introduciremos las tecnologías que vamos a utilizar para nuestros desarrollos: Python, Qt y PySide6.

## 1.1. Python



Python es un lenguaje de programación multiparadigma, interpretado, multiplataforma y libre. Nació de la mano del programador holandés Guido Van Rossum, y su primera versión fue publicada en 1991.

Es un lenguaje de propósito general, es decir, no está tipificado para un fin en concreto. Para móviles se puede utilizar Kivy.

### Características:

- **De alto nivel:** próximo al lenguaje del ser humano y no al lenguaje máquina binario.
- **Interpretado:** se ejecuta en cualquier máquina que tenga un intérprete de Python. Esto supone una gran ventaja a la hora de hacer pequeños cambios de forma rápida, ya que elimina la necesidad de recompilar el código.
- **Multiparadigma:** podemos usar la programación modular, la estructurada o la orientación a objetos según nuestras necesidades.
- **Multiplataforma:** permite que el código sea ejecutado en diferentes sistemas operativos.

- **Libre:** es propiedad de la Python Software Foundation y está publicado bajo licencia PSF-License, que es compatible con GPL (General Public License), lo que significa que es de libre uso y distribución, incluso para uso comercial.
- **Limpio y legible:** hace hincapié en la legibilidad, cosa que lo hace fácilmente comprensible y fácil de aprender. Si ya has trabajado con cualquier otro lenguaje de programación, te resultará fácil el uso de Python.
- **Tipado fuerte y dinámico:** aunque las variables son de un tipo concreto, no tenemos la necesidad de declararlas, sino que la asignación de tipos se hará en tiempo de ejecución.
- **Amplia comunidad:** gracias a su popularidad, cuenta con un amplio respaldo y se puede encontrar fácilmente mucha documentación, eventos, conferencias, etc.

Actualmente ocupa el primer sitio en el ranking **TIOBE**, que es un prestigioso indicador de la popularidad de los lenguajes de programación y se actualiza una vez al mes. No solamente eso, sino que además muestra una tendencia creciente frente a lenguajes como C o Java, que muestran la tendencia contraria. Esto se debe en gran medida a su uso mayoritario en campos como la Inteligencia Artificial, el Big Data, el Machine Learning o la Ciberseguridad, áreas predominantes en un futuro próximo.

Es importante señalar que el índice TIOBE no trata sobre el mejor lenguaje de programación o el lenguaje en el que se han escrito la mayoría de las líneas de código. El índice se calcula a partir de los resultados que se obtienen al buscar en los motores de búsqueda Google, Google Blogs, MSN, Yahoo!, Baidu, Wikipedia y YouTube. Se actualiza una vez al mes.

Todo esto nos ha llevado a escoger este lenguaje de programación para el presente módulo.

En el ejemplo siguiente se pretende comparar Python con otro lenguaje de programación para poder observar la claridad y el nivel de dificultad de programación en uno y otro. Veamos la comparación entre el «Hola Mundo» de Java y el de Python:

### Ejemplo de “Hola Mundo” en Java y Python

```
1 // Java
2 public class HolaMundo {
3     public static void main(String[] args) {
4         System.out.println("Hola Mundo!");
5     }
6 }
```

```
1 # Python
2 print("Hola mundo!")
```

Hay dos versiones de Python no compatibles entre ellas: la versión 2 y la versión 3.

Nosotros utilizaremos la versión 3 de Python. La versión 2 ha dejado de recibir soporte, y la mayoría de librerías ya poseen una implementación compatible con la versión 3.

## 1.2. Qt



Qt es un *framework* de desarrollo de aplicaciones multiplataforma para escritorio, sistemas empotrados y sistemas móviles. Sus desarrollos permiten ejecutarse en plataformas como Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS y otras.

No es un lenguaje de programación, sino un conjunto de herramientas para el desarrollo de interfaces gráficas de usuario multiplataforma mediante C++.

El desarrollo de Qt fue iniciado en 1990 por los programadores noruegos Eirik Chambe-Eng y Haavard Nord. Su empresa, Trolltech, que vendía licencias de Qt y daba soporte digital, pasó por varias adquisiciones a lo largo de los años. Hoy la antigua Trolltech se llama **The Qt Company**. Aunque The Qt Company es el principal impulsor de Qt, ahora Qt es desarrollado por un conjunto de compañías más grande: **The Qt Project**, formado por muchas empresas y personas de todo el mundo y que sigue un modelo de gobierno meritocrático.

Todos los que quieran, particulares o empresas, pueden sumarse al proyecto colaborativo escribiendo código o documentación, informando de errores o manteniendo páginas en su wiki.

Qt está disponible bajo varias licencias: The Qt Company vende licencias comerciales, pero Qt también está disponible como software libre bajo versiones de GPL y LGPL.

### Ejemplos de aplicaciones desarrolladas con Qt:

- Adobe Photoshop Album, para organizar imágenes.
- El escritorio KDE, en muchas distribuciones Linux.
- Last.fm Player, cliente de escritorio para *streaming* de música y radio.
- Skype, para mensajería y VoIP.
- TeamSpeak, para comunicación por voz, muy usada en videojuegos.
- VirtualBox, para virtualización de sistemas.
- LibreOffice, paquete ofimático libre. Alternativa a Microsoft Office.
- OnlyOffice y OnlyDesktops, competidores de LibreOffice.

## 1.3. PySide6



PySide es la unión de Python y Qt. Se trata del *binding* oficial, es decir, la implementación para poder hacer uso desde Python del framework Qt, escrito en C++. Python posee otras librerías sobre las que poder desarrollar GUIs, como son Tkinter, Kivy o wxPython.

Fue desarrollado por **The Qt Company** como parte del proyecto Qt for Python. Es una de las alternativas al paquete estándar Tkinter de Python. Al igual que Qt, PySide es software libre.

PySide es compatible con Linux/X11, macOS y Microsoft Windows, por lo que nuestros desarrollos van a ser compatibles con cualquiera de estas plataformas con un solo desarrollo de código.

Aunque existe documentación específica de PySide disponible, también podemos y recomendamos utilizar la documentación de Qt para C++, teniendo en cuenta que deberá traducirse la sintaxis de objetos y métodos C++ para adaptarlo a Python.

### **Ha habido tres versiones principales de PySide:**

- PySide: compatible con Qt 4
- PySide2: compatible con Qt 5
- PySide6: compatible con Qt 6

La versión 1 de PySide fue lanzada en agosto de 2009 bajo licencia LGPL por **Nokia**, entonces propietaria de Qt, después de no llegar a un acuerdo con los desarrolladores de PyQt (Riverbank Computing). Apoyó Qt 4 bajo los sistemas operativos Linux/X11, Mac OS X, Microsoft Windows, Maemo y MeeGo, mientras que la comunidad PySide añadió soporte para Android.

Christian Tismer inició PySide2 para llevar PySide de Qt 4 a Qt 5 en 2015. Entonces, el proyecto se incorporó al proyecto Qt. Fue lanzado en diciembre de 2018.

PySide6 se lanzó en diciembre de 2020. Añadió soporte para Qt 6 y eliminó el soporte para todas las versiones de Python anteriores a la 3.6.

**Nosotros usaremos PySide6 durante el presente curso.**

- Caso práctico - Preparar el entorno para trabajar con Python

## **2. Componentes de uso común**

En este apartado veremos cómo las aplicaciones están compuestas por diferentes componentes o controles que formarán nuestra interfaz. Además, los componentes pueden emitir señales al producirse un evento. Estas señales serán recibidas por sus escuchadores, y estos se ejecutarán al recibirlas.

## 2.1. Componentes software

En el desarrollo de circuitos electrónicos el grado de reutilización de componentes es muy alto; la construcción de un circuito electrónico se limita a la integración y el ensamblado de diferentes componentes comerciales. Por ejemplo, una placa base contiene componentes como resistencias, condensadores, relés, chips..., que no han sido fabricados por el fabricante de la placa. Así pues, existen fabricantes especializados en componentes y otros en producto final. Esto permite reducir costes en lo que a tiempo y dinero se refiere.

En el desarrollo de software la reutilización de código sigue siendo relativamente escasa, pero con la utilización de componentes este problema se reduce y, a su vez, se reduce el tiempo de desarrollo, el coste económico y los errores de programación.

El desarrollo de la interfaz de una aplicación se basa en la construcción de una aplicación a partir de componentes software ya existentes, limitando al mínimo necesario el desarrollo de código nuevo. Podemos imaginar los componentes como las piezas de un Lego que podemos conectar entre ellos en una jerarquía de árbol de componentes. Cualquier aplicación puede estar compuesta por múltiples componentes, y los componentes principales tienen componentes secundarios anidados dentro de ellos.

Anteriormente vimos una lista de los principales controles de usuario. En esta unidad pondremos en práctica el uso de dichos controles o componentes con PySide6.

## 2.2. Mi primera aplicación

Vamos a crear nuestra primera aplicación. Para empezar, crea un nuevo archivo de Python – puedes llamarlo como quieras (por ejemplo, `myapp.py`) y guárdalo en un lugar accesible. Escribiremos nuestra sencilla aplicación en este archivo.

Estaremos editando este archivo a medida que avancemos, y quizás quieras volver a versiones anteriores de tu código de vez en cuando, así que recuerda hacer copias de seguridad con regularidad.

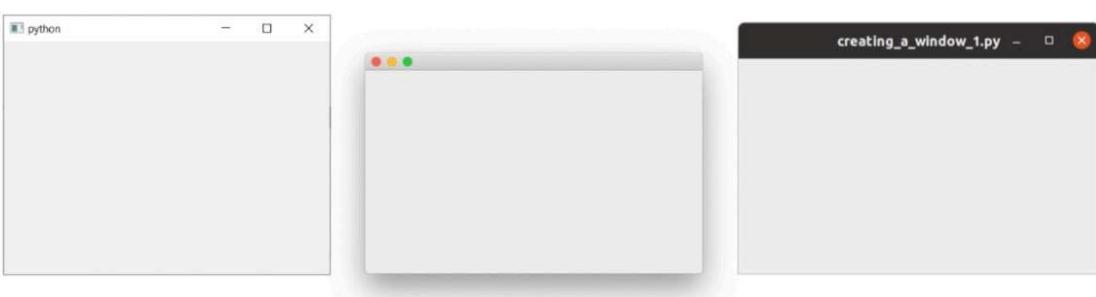
## Creando tu aplicación

El código fuente de tu primera aplicación aparece a continuación. Escríbelo tal cual y ten cuidado de no cometer errores. Si te equivocas, Python te dirá qué ha fallado.

```
1 from PySide6.QtWidgets import QApplication, QWidget
2
3 # Solo es necesario si quieres acceder a los argumentos de la línea de comando
4 import sys
5
6 # Necesitas una (y solo una) instancia de QApplication por aplicación
7 # Pasa sys.argv para permitir argumentos desde la línea de comando
8 # Si sabes que no vas a usar argumentos, QApplication([]) también es válido
9 app = QApplication(sys.argv)
10
11 # Crea un widget de Qt, que será nuestra ventana.
12 window = QWidget()
13 window.show() # IMPORTANTE: las ventanas están ocultas por defecto
14
15 # Inicia el bucle de eventos.
16 app.exec()
17
18 # La ejecución no continuará hasta que salgas y el bucle de eventos finalice
```

 **Ejecuta tu aplicación.** Ahora deberías ver tu ventana. Qt crea automáticamente una ventana con la decoración de ventana habitual, y puedes arrastrarla y cambiarle el tamaño como cualquier otra.

Lo que veas dependerá de la plataforma en la que estés ejecutando el ejemplo. La imagen de abajo muestra la ventana en Windows, macOS y Linux (Ubuntu).



## Paso a paso por el código

Vamos a repasar el código línea por línea para entender exactamente qué está ocurriendo.

Primero, importamos las clases de PySide6 que necesitamos para la aplicación. Aquí importamos `QApplication`, que maneja la aplicación, y `QWidget`, un widget gráfico básico y vacío, ambos desde el módulo `QtWidgets`.

```
1 from PySide6.QtWidgets import QApplication, QWidget
```

Los módulos principales de Qt son `QtWidgets`, `QtGui` y `QtCore`.

Podrías usar `from <módulo> import *`, pero este tipo de importaciones globales no se recomienda en Python, así que no lo usaremos aquí.

A continuación, creamos una instancia de `QApplication`, pasando `sys.argv`, que es una lista de Python que contiene los argumentos de línea de comandos pasados a la aplicación.

```
1 app = QApplication(sys.argv)
```

Si sabes que no vas a usar argumentos de línea de comandos para controlar Qt, puedes pasar una lista vacía en su lugar:

```
1 app = QApplication([])
```

Después creamos una instancia de `QWidget` usando la variable `window`.

```
1 window = QWidget()  
2 window.show()
```

En Qt, todos los widgets de nivel superior son ventanas – es decir, no tienen un *parent* y no están anidados dentro de otro widget o diseño. Esto significa que, técnicamente, puedes crear una ventana a partir de cualquier widget que quieras.

► **En detalle:**

**⚠️ No veo mi ventana**

Los widgets sin un *parent* son invisibles por defecto. Así que, después de crear el objeto `window`, **debemos llamar siempre** a `.show()` para hacerla visible. Puedes quitar la llamada a `.show()` y ejecutar la aplicación, pero no tendrás ninguna forma de salir de ella.

**¿Qué es una ventana?**

- Contiene la interfaz de usuario de tu aplicación.
- Cada aplicación necesita al menos una (aunque puede tener más).
- La aplicación, por defecto, se cierra cuando la última ventana se cierra.

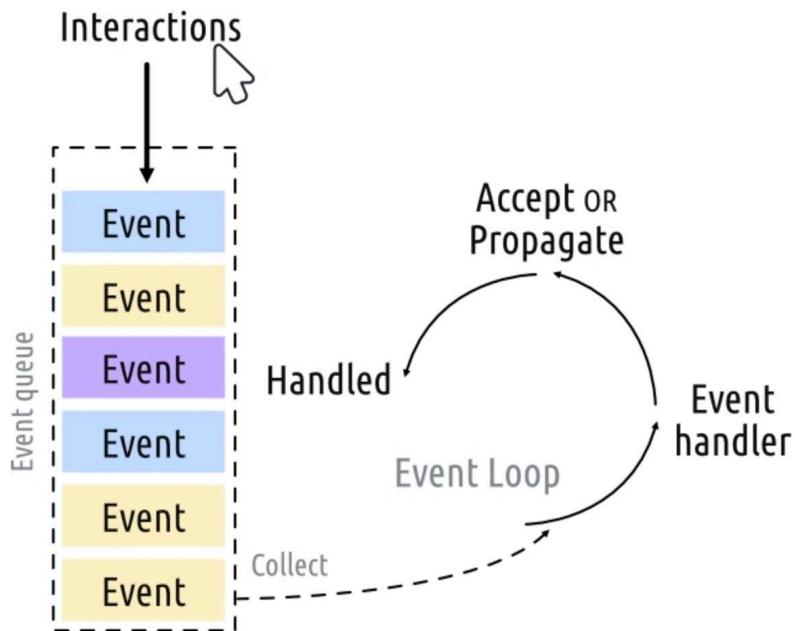
Finalmente, llamamos a `app.exec()` para iniciar el bucle de eventos.

```
1 app.exec()
```

**¿Qué es el bucle de eventos?**

Antes de mostrar la ventana en pantalla, conviene introducir algunos conceptos clave sobre cómo se organizan las aplicaciones en el mundo Qt.

El núcleo de toda aplicación Qt es la clase `QApplication`. Cada aplicación necesita –y solo necesita– un objeto `QApplication` para funcionar. Este objeto contiene **el bucle de eventos** de tu aplicación: el bucle central que gobierna toda la interacción del usuario con la interfaz gráfica.



Cada interacción con tu aplicación – ya sea una pulsación de tecla, un clic de ratón o un movimiento del cursor – genera un **evento** que se coloca en la **cola de eventos** (*event queue*).

En el bucle de eventos, la cola se revisa en cada iteración y, si se encuentra un evento en espera, este se pasa al **manejador de eventos** (*event handler*) específico.

El manejador procesa el evento y luego devuelve el control al bucle de eventos para esperar más eventos.

Solo puede haber **un bucle de eventos activo por aplicación**.

### La clase `QApplication`

- `QApplication` contiene el bucle de eventos de Qt
- Se requiere una única instancia de `QApplication`
- Tu aplicación permanece “en espera” en el bucle de eventos hasta que se realiza alguna acción

- Solo puede existir un único bucle de eventos por aplicación

## QMainWindow

Como vimos anteriormente, en Qt cualquier widget puede ser una ventana. Por ejemplo, si reemplazas `QWidget` por `QPushButton`, obtendrás una ventana con un único botón que se puede pulsar:

```
1 import sys
2
3 from PySide6.QtWidgets import QApplication, QPushButton
4
5 app = QApplication(sys.argv)
6
7 window = QPushButton("Push Me")
8 window.show()
9
10 app.exec()
```

Esto está bien, aunque no es especialmente útil: es raro que necesites una interfaz que consista en un único control. Más adelante veremos que gracias al uso de **layouts** se pueden anidar widgets dentro de otros para construir interfaces más complejas dentro de un `QWidget`.

Sin embargo, Qt ya ofrece una solución lista para usar: `QMainWindow`.

Este es un widget predefinido que proporciona muchas de las características estándar de las ventanas que vas a necesitar en tus aplicaciones, como barras de herramientas, menús, barras de estado, widgets acoplables y más.

Veremos esas funciones avanzadas más adelante; por ahora, añadiremos una `QMainWindow` vacía a nuestra aplicación.

```
1 from PySide6.QtWidgets import QApplication, QMainWindow
2 import sys
3
4 app = QApplication(sys.argv)
5
6 window = QMainWindow()
7 window.show() # IMPORTANTE: las ventanas están ocultas por defecto
8
9 # Inicia el bucle de eventos.
10 app.exec()
```

■ **Ejecuta tu aplicación.** Ahora deberías ver tu ventana principal. Visualmente se verá igual que antes.

Nuestra `QMainWindow` no es muy interesante todavía. Podemos arreglarlo añadiendo contenido.

Si quieras crear una ventana personalizada, la mejor forma es **crear una subclase de `QMainWindow`** y definir toda la configuración de la ventana dentro del bloque `__init__`.

Esto permite que el comportamiento de la ventana esté bien encapsulado.

Vamos a crear nuestra propia subclase llamada `MainWindow` para mantenerlo simple.

```
1 import sys
2
3 from PySide6.QtWidgets import (
4     QApplication,
5     QMainWindow,
6     QPushButton,
7 )
8
9 # Subclase de QMainWindow para personalizar la ventana principal
10 class MainWindow(QMainWindow):
11     def __init__(self):
12         super().__init__()
13
14         self.setWindowTitle("My App")
15
16         button = QPushButton("Press Me!")
17
18         # Establece el widget central de la ventana.
19         self.setCentralWidget(button)
20
21 app = QApplication(sys.argv)
22
23 window = MainWindow()
24 window.show()
25
26 app.exec()
```

## ✓ Notas importantes

- Los widgets comunes de Qt se importan siempre desde el módulo `QtWidgets`.
- Siempre debemos llamar al método `__init__` de la superclase (`super().__init__()`).
- Se usa `.setCentralWidget()` para colocar un widget en el centro de la `QMainWindow`.

 Cuando haces una subclase de una clase de Qt, **debes llamar siempre a `super().__init__()`** para permitir que Qt configure correctamente el objeto.

En nuestro bloque `__init__`, primero usamos `.setWindowTitle()` para cambiar el título de la ventana principal.

Después añadimos nuestro primer widget – un `QPushButton` – en el centro de la ventana. Este es uno de los widgets básicos disponibles en Qt.

Al crear el botón puedes pasar el texto que quieras que muestre.

Finalmente, llamamos a `.setCentralWidget()` en la ventana. Esta es una función específica de `QMainWindow` que permite establecer el widget que se colocará en el centro.

■ **Ejecuta tu aplicación.** Volverás a ver la ventana, pero ahora con un botón `QPushButton` en el centro. De momento, al pulsarlo no hará nada – eso será lo siguiente que resolveremos.



### 🍴 ¿Con ganas de widgets?

Pronto veremos más widgets en detalle, pero si no puedes esperar y quieres adelantarte, puedes echar un vistazo a la [documentación de QWidget](#).

Prueba a añadir diferentes widgets a tu ventana.

### Redimensionar ventanas y widgets

La ventana actualmente se puede redimensionar libremente: si arrastras cualquier esquina con el ratón puedes cambiarla a cualquier tamaño.

Aunque está bien permitir que los usuarios ajusten el tamaño de tus aplicaciones, en ocasiones puede que quieras **restringir el tamaño mínimo o máximo, o bloquear la ventana a un tamaño fijo**.

En Qt, los tamaños se definen usando un objeto `QSize`, que acepta como parámetros **ancho** y **alto**, en ese orden. Cuando queremos establecer un tamaño fijo en una ventana u otro widget, muchos métodos de Qt esperan recibir un objeto `QSize`, no dos números sueltos.

Por ejemplo, el siguiente código creará una ventana de tamaño fijo de **400x300 píxeles**:

```
1 import sys
2
3 from PySide6.QtCore import QSize, Qt
4 from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
5
6 # Subclase de QMainWindow para personalizar la ventana principal
7 class MainWindow(QMainWindow):
8     def __init__(self):
9         super().__init__()
10
11         self.setWindowTitle("My App")
12
13         button = QPushButton("Press Me!")
14
15         # Establecer un tamaño fijo para la ventana
16         self.setFixedSize(QSize(400, 300))
17
18         # Establecer el widget central de la ventana
19         self.setCentralWidget(button)
20
21 app = QApplication(sys.argv)
22
23 window = MainWindow()
24 window.show()
25
26 app.exec()
```

### Configuración del tamaño de la ventana

 **Ejecuta tu aplicación.** Verás una ventana con un tamaño fijo. Intenta redimensionarla: no funcionará.



(Imágenes de la ventana de tamaño fijo en Windows, macOS y Linux. En Windows y Linux el botón de maximizar está deshabilitado; en macOS se puede maximizar pero el widget central no cambia de tamaño).

Además de `.setFixedSize()`, también puedes usar:

- `.setMinimumSize()` → para establecer un tamaño mínimo
- `.setMaximumSize()` → para establecer un tamaño máximo

Puedes experimentar con esto tú mismo.

**Puedes usar estos métodos de tamaño en cualquier widget.**

En esta sección hemos visto:

- La clase `QApplication`
- La clase  `QMainWindow`
- El bucle de eventos
- Cómo añadir un widget simple a una ventana
- Cómo establecer restricciones de tamaño

En la próxima sección veremos los mecanismos que Qt proporciona para que los widgets y las ventanas se comuniquen entre sí y con tu propio código.

**Guarda una copia de tu archivo como `myapp.py`, ya que lo necesitaremos más adelante.**

#### ► Ejercicio propuesto 1

## 2.3. Señales y slots

En las aplicaciones con interfaz gráfica (GUI), los widgets deben poder responder a las acciones del usuario: un clic, un cambio de texto, mover un control deslizante...

Para que eso ocurra, Qt utiliza un mecanismo propio basado en **señales (signals)** y **slots**. Este sistema permite que diferentes partes de una aplicación se comuniquen entre sí sin depender directamente unas de otras.

Podemos pensar en ello como una relación **acción → reacción**:

cada vez que ocurre un evento, Qt emite una señal; si hay una función conectada a esa señal, se ejecuta como respuesta.

### ¿Qué son las señales?

Las **señales** son notificaciones emitidas por los widgets cuando sucede algo. Ese "algo" puede ser casi cualquier evento: pulsar un botón, cambiar el texto de una caja de entrada o incluso modificar el título de una ventana.

Aunque muchas señales se activan por interacción del usuario, también pueden hacerlo desde el propio código.

Además de avisar de que algo ha pasado, las señales pueden **enviar datos** para dar más contexto sobre el evento.

💡 También puedes definir tus **propias señales personalizadas**, algo que veremos más adelante.

### ¿Qué son los slots?

Los **slots** son los receptores de las señales. En Python, cualquier función o método puede actuar como slot: basta con conectarlo a una señal.

Si la señal envía datos, el slot los recibirá como parámetros.

Muchos widgets de Qt ya tienen sus propios slots integrados, lo que permite conectar widgets entre sí directamente sin necesidad de escribir código adicional.

Veamos ahora los fundamentos de las señales en Qt y cómo puedes utilizarlas para conectar widgets y hacer que tu aplicación reaccione a eventos.



Widget	Señales (ejemplos)	Funciones útiles (ejemplos)
<b>QCheckBox</b>	<code>stateChanged(int)</code> , <code>toggled(bool)</code>	<code>isChecked()</code> , <code>setChecked(bool)</code> , <code>setCheckState(Qt.CheckState)</code>
<b>QLabel</b>	(no “de cambio de texto”) <code>linkActivated(str)</code> , <code>linkHovered(str)</code>	<code>setText(str)</code>
<b>QGroupBox</b>	<code>toggled(bool)</code> (si es checkable)	<code>setCheckable(bool)</code> , <code>setChecked(bool)</code> , <code>isChecked()</code> , <code>setTitle(str)</code>
<b>QComboBox</b>	<code>currentIndexChange d(int/str)</code> , <code>currentTextChanged(str)</code> , <code>activated(int)</code>	<code>addItem(str)</code> , <code>addItems(list)</code> , <code>setCurrentIndex(index)</code> , <code>setCurrentText(str)</code>
<b>QRadioButton</b>	<code>toggled(bool)</code> , <code>clicked(bool)</code>	<code>isChecked()</code> , <code>setChecked(bool)</code>
<b>QPushButton</b>	<code>clicked(bool)</code> , <code>pressed()</code> , <code>released()</code> , <code>toggled(bool)</code>	<code>setCheckable(bool)</code> , <code>setChecked(bool)</code> , <code>isChecked()</code>
<b>QTabWidget</b>	<code>currentChanged(int)</code> , <code>tabCloseRequested(int)</code>	<code>addTab(QWidget, str)</code> , <code>setCurrentWidget(QWidget)</code> , <code>setCurrentIndex(index)</code> , <code>removeTab(int)</code>
<b>QTableWidget</b>	<code>cellChanged(int, int)</code> , <code>currentCellChanged(int, int, int, int)</code> , <code>itemChanged(QTable</code>	<code>insertRow(int)</code> , <code>removeRow(int)</code> , <code>setItem(int, int, QTableWidgetItem)</code> , <code>setRowCount(int)</code>

WidgetItem*)		
<b>QLineEdit</b>	<code>textChanged(str)</code> , <code>textEdited(str)</code> , <code>returnPressed()</code>	<code>setText(str)</code> , <code>clear()</code> , <code>setPlaceholderText(str)</code>
<b>QTextEdit</b>	<code>textChanged()</code>	<code>setPlainText(str)</code> , <code>setHtml(str)</code> , <code>clear()</code>
<b>QProgressBar</b>	<code>valueChanged(int)</code>	<code>setValue(int)</code> , <code>setMaximum(int)</code> , <code>setMinimum(int)</code>
<b>QDateTimeEdit</b>	<code>dateChanged(QDate)</code> , <code>dateTimeChanged(QDateTime)</code> , <code>timeChanged(QTime)</code>	<code> setDate(QDate)</code> , <code>setDateTime(QDateTime)</code> , <code> setTime(QTime)</code>
<b>QSlider</b>	<code>valueChanged(int)</code> , <code>sliderMoved(int)</code> , <code>sliderPressed()</code>	<code>setValue(int)</code> , <code>setMinimum(int)</code> , <code>setMaximum(int)</code> , <code>setRange(int, int)</code>
<b>QDial</b>	<code>valueChanged(int)</code>	<code>setValue(int)</code> , <code>setMinimum(int)</code> , <code>setMaximum(int)</code> , <code>setRange(int, int)</code>

💡 Carga una copia nueva de `myapp.py` y guárdala con otro nombre para esta sección.

### Señales de QPushButton

Nuestra aplicación sencilla tiene actualmente un **QMainWindow** con un **QPushButton** como widget central. Vamos a conectar este botón a un método Python personalizado que actuará como slot.

Aquí creamos un slot sencillo llamado `the_button_was_clicked`, que recibirá la señal **clicked** del **QPushButton**.

```
1  from PySide6.QtWidgets import (
2      QApplication,
3      QMainWindow,
4      QPushButton,
5  )
6  import sys
7
8  class MainWindow(QMainWindow):
9      def __init__(self):
10          super().__init__() # Llamada al constructor de la super
11
12          self.setWindowTitle("My App")
13
14          button = QPushButton("Press Me!")
15          button.clicked.connect(self.the_button_was_clicked)
16
17          # Establecer el widget central de la ventana
18          self.setCentralWidget(button)
19
20      def the_button_was_clicked(self):
21          print("Clicked!")
22
23  app = QApplication(sys.argv)
24
25  window = MainWindow()
26  window.show()
27
28  app.exec()
```

### ► Ejecuta el programa.

Si haces clic en el botón, verás el texto **Clicked!** en la consola.

#### Salida en consola:

```
1 Clicked!
2 Clicked!
3 Clicked!
4 Clicked!
```

Elemento	Qué hace	Ejemplo
Señal	Notifica que ha ocurrido un evento	button.clicked
Slot	Responde a la señal	def the_button_was_clicked():
Conexión	Une señal y slot	button.clicked.connect(...)

## Recepción de datos

Hasta ahora hemos visto que las señales pueden **enviar información adicional** junto con el aviso de que algo ha ocurrido.

Esto permite que el *slot* reciba más contexto sobre el evento y pueda reaccionar de forma distinta según los datos recibidos.

La señal `.clicked` de un `QPushButton`, por ejemplo, no solo indica que el botón ha sido pulsado, sino que también puede enviar un valor booleano llamado `checked`, que representa si el botón ha quedado **activado o desactivado** tras el clic.

En los botones normales este valor siempre es `False`, porque no mantienen estado.

Pero si convertimos el botón en **comutable** con `setCheckable(True)`, ese valor alternará entre `True` y `False` cada vez que se pulse.

En el siguiente ejemplo añadimos un segundo *slot* para mostrar el valor `checked` y comprobar cómo varía con cada clic.

```
1 import sys
2 from PySide6.QtWidgets import (
3     QApplication,
4     QMainWindow,
5     QPushButton,
6 )
7
8 class MainWindow(QMainWindow):
9     def __init__(self):
10         super().__init__() # Llamada al constructor de la super
11
12         self.setWindowTitle("My App")
13
14         button = QPushButton("Press Me!")
15         button.setCheckable(True)
16
17         # Conectar la misma señal a dos slots distintos
18         button.clicked.connect(self.the_button_was_clicked)
19         button.clicked.connect(self.the_button_was_toggled)
20
21         # Establecer el widget central de la ventana
22         self.setCentralWidget(button)
23
24     def the_button_was_clicked(self):
25         print("Clicked!")
26
27     def the_button_was_toggled(self, checked):
28         print("Checked?", checked)
29
30 app = QApplication(sys.argv)
31
32 window = MainWindow()
33 window.show()
34
35 app.exec()
```

## ► Ejecuta el programa.

Si pulsas el botón, lo verás resaltado como *checked*.

Si lo pulsas de nuevo, volverá a su estado original.

Fíjate en la consola para ver el estado **check**.

### Salida en consola:

```
1 Clicked!
2 Checked? True
3 Clicked!
4 Checked? False
```

#### 💡 Nota:

Puedes conectar una misma señal a varios *slots*, o incluso varios widgets a un mismo *slot*.

Esto permite construir interfaces donde múltiples elementos reaccionan al mismo evento o coordinan su comportamiento entre sí.

- ▶ Ejercicio propuesto 2
- ▶ Ejercicio propuesto 2b

### Almacenar datos

A menudo es útil guardar el **estado actual** de un widget en una variable de Python, lo cual permite trabajar con ese valor sin tener que acceder constantemente al widget original.

Puedes guardar estos valores como variables individuales o dentro de un diccionario si prefieres agrupar varios estados.

En el siguiente ejemplo guardamos el valor `checked` de un botón en una variable llamada `button_is_checked`, dentro de `self`.

```
1 import sys
2 from PySide6.QtWidgets import (
3     QApplication,
4     QMainWindow,
5     QPushButton,
6 )
7
8 class MainWindow(QMainWindow):
9     def __init__(self):
10         super().__init__() # Llamada al constructor de la super
11
12         self.button_is_checked = True # ① Valor por defecto de
13
14         self.setWindowTitle("My App")
15
16         button = QPushButton("Press Me!")
17         button.setCheckable(True)
18         button.setChecked(self.button_is_checked) # ② Sincroniz
19         button.clicked.connect(self.the_button_was_toggled) # ③
20
21         self.setCentralWidget(button)
22
23     def the_button_was_toggled(self, checked):
24         self.button_is_checked = checked # ④ Guardamos el nuevo
25         print("Estado del botón:", self.button_is_checked)
26
27 app = QApplication(sys.argv)
28 window = MainWindow()
29 window.show()
30 app.exec()
```

① **Valor inicial:** definimos una variable para almacenar el estado del botón.

② **Sincronización inicial:** el botón muestra ese valor desde el inicio.

③ **Actualización dinámica:** cada vez que cambia el estado, la variable se actualiza automáticamente.

 **Sugerencia:** si tienes varios botones o controles, puedes usar un diccionario:

```
1 self.estados = {"principal": False, "secundario": True}
```

Así puedes acceder fácilmente al estado de cada uno sin crear demasiadas variables separadas.

Puedes usar este mismo patrón con cualquier widget de PySide6.

Si un widget no proporciona una señal que envíe su estado actual, puedes **consultar directamente su valor** desde el *slot* que maneje el evento.

En el siguiente ejemplo comprobamos el estado **checked** dentro del manejador `the_button_was_released`, que se ejecuta cuando el botón se suelta.

```
1 import sys
2 from PySide6.QtWidgets import (
3     QApplication,
4     QMainWindow,
5     QPushButton,
6 )
7
8 class MainWindow(QMainWindow):
9     def __init__(self):
10         super().__init__() # Llamada al constructor de la super
11
12         self.button_is_checked = True
13
14         self.setWindowTitle("My App")
15
16         self.button = QPushButton("Press Me!") # ① Guardamos re
17         self.button.setCheckable(True)
18         self.button.released.connect(
19             self.the_button_was_released # ② La señal released
20         )
21         self.button.setChecked(self.button_is_checked)
22
23         self.setCentralWidget(self.button)
24
25     def the_button_was_released(self):
26         self.button_is_checked = self.button.isChecked() # ③ Ob
27         print("Estado actual:", self.button_is_checked)
28
29
30 app = QApplication(sys.argv)
31 window = MainWindow()
32 window.show()
33 app.exec()
```

① **Referencia al botón:** la guardamos en `self` para poder acceder a él dentro del slot.

② **Señal sin datos:** `released` se emite al soltar el botón, pero no envía el estado como parámetro.

③ **Consulta directa:** con `.isChecked()` obtenemos el estado actual del botón y lo almacenamos.

 **Recuerda:** este enfoque se aplica también a otros widgets, como `QLineEdit`, `QSlider` o `QCheckBox`.

Si la señal no incluye el valor que necesitas, siempre puedes pedirlo directamente al widget.

## Cambiar la interfaz

Hasta ahora hemos visto cómo aceptar señales y mostrar información en la consola. Pero, ¿qué pasa si queremos que ocurra algo **en la interfaz** al pulsar el botón?

Vamos a actualizar nuestro slot para modificar el propio botón: cambiar su texto y desactivarlo, de modo que ya no pueda volver a pulsarse.

También eliminaremos el estado *checkable* para este ejemplo.

```
1 from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
2 import sys
3
4 class MainWindow(QMainWindow):
5     def __init__(self):
6         super().__init__()
7
8         self.setWindowTitle("My App")
9
10        self.button = QPushButton("Press Me!") # ① Necesitamos
11        self.button.clicked.connect(self.the_button_was_clicked)
12
13        # Establecer el widget central
14        self.setCentralWidget(self.button)
15
16    def the_button_was_clicked(self):
17        self.button.setText("You already clicked me.") # ② Cambiamos
18        self.button.setEnabled(False) # ③ Desactivamos
19        self.setWindowTitle("My Oneshot App") # También
20
21 app = QApplication(sys.argv)
22
23 window = MainWindow()
24 window.show()
25
26 app.exec()
```

① Guardamos una referencia al botón para poder acceder a él en el método `the_button_was_clicked`.

② Cambiamos el texto pasando una cadena a `.setText()`.

③ Para desactivar un botón usamos `.setEnabled(False)`.

### ▶ Ejecuta el programa.

Al hacer clic, el texto cambiará y el botón quedará desactivado.

No estás limitado a modificar solo el botón que emite la señal; puedes hacer **cualquier cosa** dentro de tus métodos de slot.

Por ejemplo, añade la siguiente línea dentro de `the_button_was_clicked` para cambiar también el título de la ventana:

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

La mayoría de widgets tienen sus propias señales, y el `QMainWindow` que estamos usando no es una excepción.

En el siguiente ejemplo más complejo, conectaremos la señal `.windowTitleChanged` del `QMainWindow` a un método slot personalizado llamado `the_window_title_changed`.

Este slot también recibe el nuevo título de la ventana.

```
1  from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
2  import sys
3  from random import choice
4
5  window_titles = [ # ① Lista de títulos entre los que elegiremos
6      "My App",
7      "My App",
8      "My App",
9      "Still My App",
10     "Still My App",
11     "What on earth",
12     "What on earth",
13     "This is surprising",
14     "This is surprising",
15     "Something went wrong",
16 ]
17
18 class MainWindow(QMainWindow):
19     def __init__(self):
20         super().__init__()
21
22         self.n_times_clicked = 0
23
24         self.setWindowTitle("My App")
25
26         self.button = QPushButton("Press Me!")
27         self.button.clicked.connect(self.the_button_was_clicked)
28
29         # ② Conectamos nuestro método personalizado al evento
30         self.windowTitleChanged.connect(
31             self.the_window_title_changed
32         )
33
34         # Establecer el widget central
35         self.setCentralWidget(self.button)
36
37     def the_button_was_clicked(self):
38         print("Clicked.")
39         new_window_title = choice(window_titles)
40         print("Setting title: %s" % new_window_title)
41         self.setWindowTitle(new_window_title) # ③ Cambiar el
42
43     def the_window_title_changed(self, window_title):
```

```
44     print("Window title changed: %s" % window_title) # ④
45     if window_title == "Something went wrong":
46         self.button.setDisabled(True) # Desactivar el botón
47
48 app = QApplication(sys.argv)
49
50 window = MainWindow()
51 window.show()
52
53 app.exec()
```

- ① Lista de posibles títulos para la ventana, seleccionados aleatoriamente con `random.choice()`.
- ② Conectamos el método `the_window_title_changed` a la señal `.windowTitleChanged` de la ventana.
- ③ Cambiamos el título de la ventana al nuevo valor.
- ④ Mostramos en consola el título actualizado.

### ► Ejecuta el programa.

Haz clic varias veces en el botón hasta que el título cambie a "Something went wrong". Cuando ocurra, el botón se desactivará.

Hay algunos detalles a tener en cuenta:

- La señal `windowTitleChanged` **no se emite siempre** al establecer el título de la ventana. Solo se emite si el nuevo título es diferente del anterior. Si se intenta establecer el mismo título repetidamente, la señal solo se lanza la primera vez.
- Es importante revisar bien las condiciones bajo las que las señales se activan, para evitar sorpresas al usarlas en tu aplicación.

También conviene observar cómo se pueden **encadenar efectos** usando señales.

Un único suceso (por ejemplo, pulsar un botón) puede desencadenar varios efectos en cascada.

Estos efectos no necesitan saber qué los provocó; simplemente ocurren como resultado de reglas simples.

Esta separación entre causa y efecto es clave cuando construyes aplicaciones con interfaz gráfica.

En esta sección hemos visto señales y slots: cómo usarlos para pasar datos y estado entre distintas partes de tu aplicación.

A continuación veremos cómo **conectar widgets directamente entre sí**.

## Conexión directa entre widgets

 *Nota:* En el siguiente ejemplo usamos un **layout** (`QVBoxLayout`) para organizar los widgets. No te preocupes si aún no sabes cómo funcionan; lo explicaremos con detalle en la próxima sección.

Hasta ahora hemos visto ejemplos de conectar señales de widgets a métodos de Python. Cuando se lanza una señal, el método se ejecuta y recibe los datos.

Pero no siempre es necesario usar un método intermedio: también puedes conectar **directamente** widgets entre sí.

En el siguiente ejemplo añadimos un **QLineEdit** y un **QLabel** a la ventana.

En el `__init__` conectamos la señal `textChanged` del **QLineEdit** al método `setText` del **QLabel**.

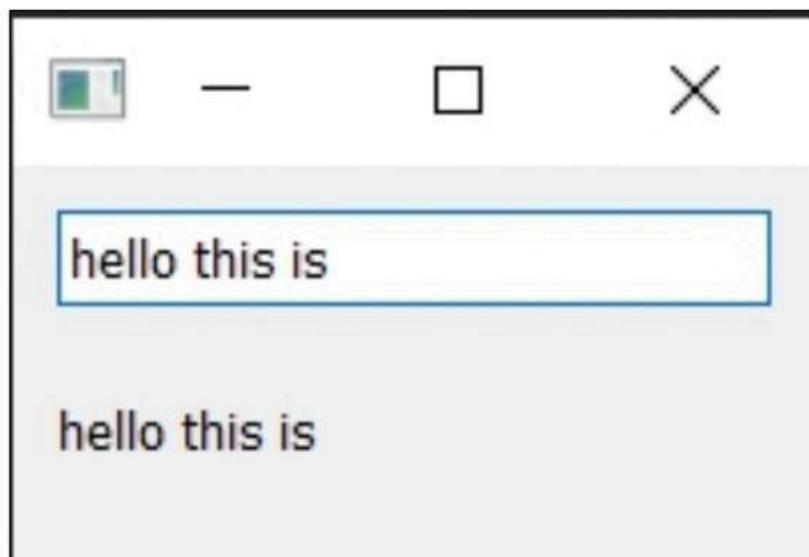
Cada vez que cambie el texto en el **QLineEdit**, el **QLabel** se actualizará automáticamente.

```
1  from PySide6.QtWidgets import (
2      QApplication,
3      QMainWindow,
4      QLabel,
5      QLineEdit,
6      QVBoxLayout,
7      QWidget,
8  )
9  import sys
10
11 class MainWindow(QMainWindow):
12     def __init__(self):
13         super().__init__()
14
15         self.setWindowTitle("My App")
16
17         self.label = QLabel()
18         self.input = QLineEdit()
19
20         # ① Conectar directamente la señal textChanged del input
21         self.input.textChanged.connect(self.label.setText)
22
23         # ② Añadimos los widgets a un layout vertical
24         layout = QVBoxLayout()
25         layout.addWidget(self.input)
26         layout.addWidget(self.label)
27
28         container = QWidget()
29         container.setLayout(layout)
30
31         # Establecer el widget central
32         self.setCentralWidget(container)
33
34 app = QApplication(sys.argv)
35
36 window = MainWindow()
37 window.show()
38
39 app.exec()
```

- ① Para conectar el input al label, ambos deben estar definidos previamente.

② Este código añade ambos widgets a un layout vertical y lo establece como el contenido principal de la ventana (veremos los layouts en detalle más adelante).

▶ ¡Ejecuta el programa! Escribe algo de texto en la caja superior y verás cómo aparece inmediatamente en la etiqueta.



Cualquier texto escrito en la entrada aparece inmediatamente en la etiqueta.

La mayoría de los *widgets* de Qt tienen *slots* disponibles, a los cuales puedes conectar cualquier señal (*signal*) que emita el mismo tipo que acepta. La documentación del *widget* incluye los *slots* de cada uno, listados bajo “Public Slots”. Por ejemplo, consulta la documentación de Qt para [QLabel](#).

En esta sección hemos cubierto las señales (*signals*) y los *slots*. Hemos mostrado algunas señales simples y cómo usarlas para pasar datos y estados dentro de tu aplicación.

A continuación, veremos los *widgets* que Qt ofrece para su uso en tus aplicaciones, junto con las señales que proporcionan.

## 2.4. Widgets

En Qt, *widget* es el nombre que se le da a un componente de la interfaz gráfica (UI) con el que el usuario puede interactuar. Las interfaces están formadas por múltiples widgets, dispuestos dentro de una ventana. Qt incluye una gran selección de widgets listos para usar, y también permite crear widgets personalizados.

Puedes ejecutar el siguiente código para mostrar una colección de widgets en una ventana.



widgets\_list.py

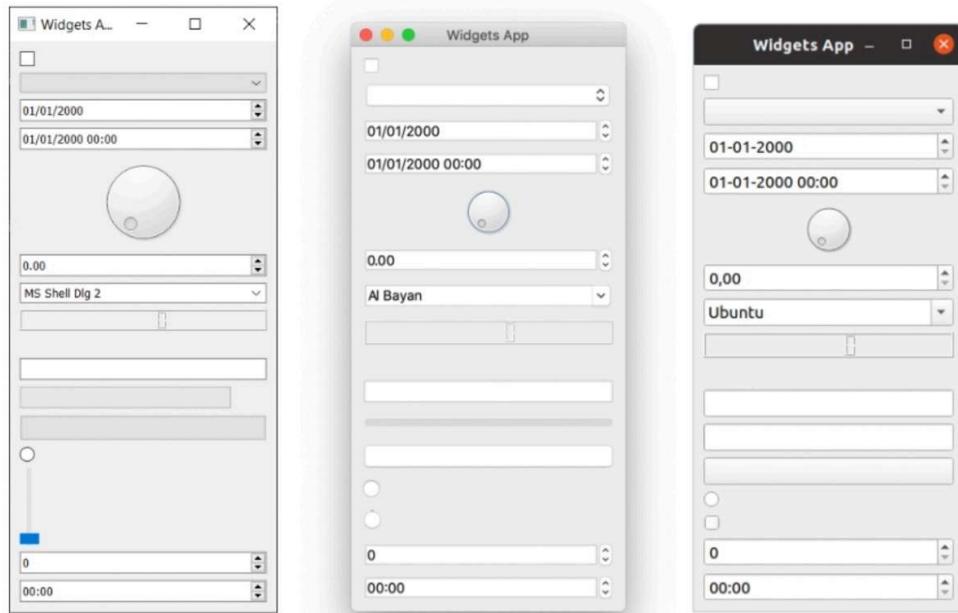
Text File

1.3 KB

Este archivo utiliza algunos trucos que se explicarán más adelante, así que no te preocupes por entender el código todavía.

### ▶ Ejecuta el programa.

Verás una ventana con múltiples widgets interactivos.



La ventana de ejemplo con widgets, mostrada en Windows, macOS y Linux (Ubuntu).

Los widgets que aparecen en el ejemplo anterior son los siguientes (de arriba a abajo):

Widget	Qué hace
QCheckbox	Una casilla de verificación
QComboBox	Lista desplegable
QDateEdit	Para editar fechas
QDateTimeEdit	Para editar fechas y horas
QDial	Dial giratorio
QDoubleSpinBox	Selector numérico para decimales
QFontComboBox	Lista de fuentes
QLCDNumber	Visualizador LCD (bastante básico)
QLabel	Solo etiqueta, no interactiva
QLineEdit	Para introducir una línea de texto
QProgressBar	Barra de progreso
QPushButton	Botón
QRadioButton	Grupo con una única opción activa
QSlider	Deslizador
QSpinBox	Selector numérico entero
QTimeEdit	Para editar horas

Hay muchos más widgets además de estos, pero no caben todos aquí.

Para la lista completa puedes consultar la [documentación de Qt](#).

En las siguientes secciones veremos algunos de los más útiles con más detalle.

 Carga una nueva copia de `myapp.py` y guárdala con otro nombre para trabajar esta sección.

## QLabel

Empezaremos con **QLabel**, uno de los widgets más simples. Sirve para mostrar texto o imágenes fijas dentro de la interfaz.

Puedes crear una etiqueta directamente con un texto:

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

O bien modificar su contenido más tarde con `.setText()`:

```
1 widget = QLabel("1") # La etiqueta se crea con el texto "1"
2 widget.setText("2") # Ahora la etiqueta muestra "2"
```

También puedes ajustar parámetros de fuente, como el tamaño o la alineación del texto.

```
1 import sys
2 from PySide6.QtCore import Qt
3 from PySide6.QtWidgets import QApplication, QLabel, QMainWindow
4
5 class MainWindow(QMainWindow):
6     def __init__(self):
7         super().__init__()
8
9         self.setWindowTitle("My App")
10
11         widget = QLabel("Hello")
12         font = widget.font()          # ① Obtenemos la fuente actual
13         font.setPointSize(30)         # Cambiamos el tamaño de la
14         widget.setFont(font)
15         widget.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
16
17         self.setCentralWidget(widget)
18
19 app = QApplication(sys.argv)
20
21 window = MainWindow()
22 window.show()
23
24 app.exec()
```

- ① Usamos `<widget>.font()` para obtener la fuente actual, la modificamos y la volvemos a aplicar. Así mantenemos la coherencia con las convenciones del sistema.

② La alineación se especifica usando flags del namespace `Qt`.

- ▶ **Ejecuta el programa.** Ajusta los parámetros de la fuente y observa el resultado.



*QLabel en Windows, macOS y Ubuntu.*

💡 El namespace `Qt` está lleno de atributos que puedes usar para personalizar y controlar los widgets.

Más adelante veremos esto con detalle en la sección **Enums & the Qt Namespace.**

## Alineación horizontal

Flag	Comportamiento
<code>Qt.AlignLeft</code>	Alinea a la izquierda
<code>Qt.AlignRight</code>	Alinea a la derecha
<code>Qt.AlignHCenter</code>	Centra horizontalmente en el espacio disponible
<code>Qt.AlignJustify</code>	Justifica el texto en el espacio disponible

## Alineación vertical

Flag	Comportamiento
<code>Qt.AlignTop</code>	Alinea en la parte superior
<code>Qt.AlignBottom</code>	Alinea en la parte inferior
<code>Qt.AlignVCenter</code>	Centra verticalmente en el espacio disponible

Puedes combinar flags utilizando el **operador OR (|)**, por ejemplo:

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

► **Ejecuta el programa.** Prueba a combinar diferentes flags de alineación y observa cómo cambia la posición del texto.

### 💡 Qt Flags

Cuando combinas flags, en realidad estás uniendo *bitmasks* no superpuestas.

Por ejemplo:

- `Qt.AlignLeft` tiene el valor binario `0b0001`
- `Qt.AlignBottom` es `0b0100`

Si los combinás con OR (|), obtienes `0b0101`, que representa "abajo a la izquierda".

También existe un flag abreviado que centra en ambas direcciones simultáneamente:

Flag	Comportamiento
<code>Qt.AlignCenter</code>	Centra horizontal y vertical simultáneamente

### ► Ejercicio propuesto 3

#### Mostrar imágenes con QLabel

Además de texto, también puedes usar `QLabel` para mostrar una **imagen** en tu aplicación. Para hacerlo, se utiliza el método `.setPixmap()`, que recibe un objeto `QPixmap`.

Un `QPixmap` es, básicamente, una imagen cargada en memoria (una matriz de píxeles). Puedes crear un `QPixmap` directamente a partir de un archivo de imagen.

```
1 import sys
2 from PySide6.QtGui import QPixmap
3 from PySide6.QtWidgets import QApplication, QLabel, QMainWindow
4
5 class MainWindow(QMainWindow):
6     def __init__(self):
7         super().__init__()
8
9         self.setWindowTitle("My App")
10
11        widget = QLabel("Hello")
12        widget.setPixmap(QPixmap("otje.jpg"))
13
14        self.setCentralWidget(widget)
15
16 app = QApplication(sys.argv)
17
18 window = MainWindow()
19 window.show()
20
21 app.exec()
```



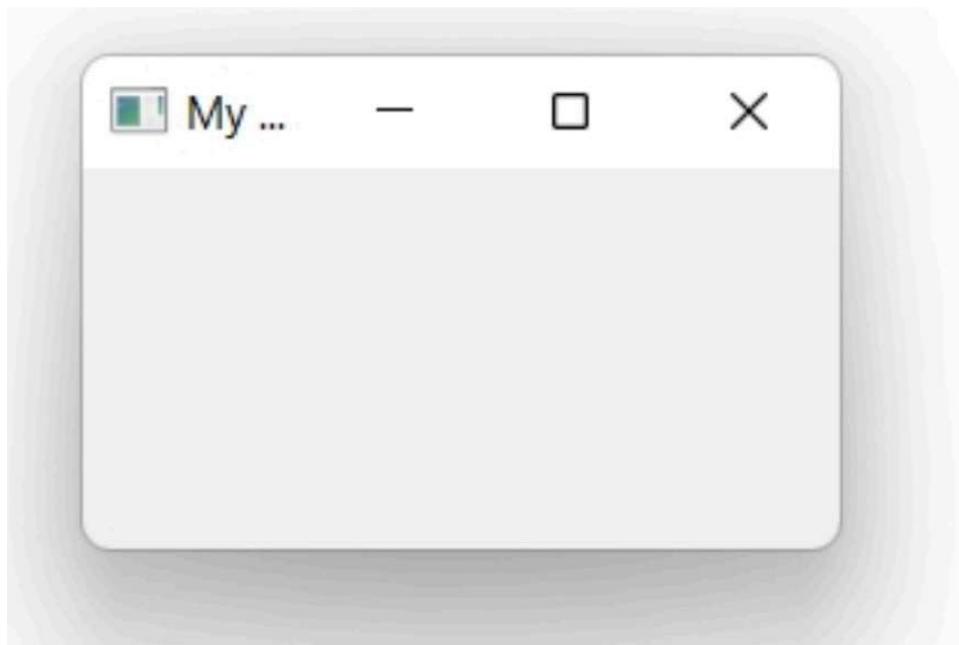
Otje el gato. Qué cara tan encantadora.

► **Ejecuta el programa.** Redimensiona la ventana: la imagen quedará rodeada por espacio vacío.

💡 ¿No se ve la imagen? Sigue leyendo.

En el ejemplo anterior hemos indicado el nombre del archivo como "otje.jpg", por lo que se cargará desde la **carpeta actual** desde la que se ejecuta la aplicación.

Sin embargo, esta carpeta **no siempre** es la misma en la que está el script. Si ejecutas el script desde otro directorio, no encontrará la imagen y no se cargará.



Otje ha desaparecido.

💡 Esto es un problema frecuente cuando se ejecutan scripts desde IDEs, que establecen las rutas en función del proyecto activo.

### Cómo asegurarte de que la ruta funcione siempre

Para evitar errores al cargar imágenes, puedes construir la ruta de forma **relativa al propio archivo del script**.

La variable especial `__file__` contiene la ruta completa del archivo Python que se está ejecutando.

Con la función `os.path.dirname()` obtenemos la carpeta donde se encuentra ese archivo, y con `os.path.join()` podemos crear rutas seguras a partir de ella.

```
1 import os
2 import sys
3 from PySide6.QtGui import QPixmap
4 from PySide6.QtWidgets import QApplication, QLabel, QMainWindow
5
6 basedir = os.path.dirname(__file__)
7 print("Current working folder:", os.getcwd()) # ① Carpeta de trabajo
8 print("Paths are relative to:", basedir) # ② Ruta base (archivo)
9
10 class MainWindow(QMainWindow):
11     def __init__(self):
12         super().__init__()
13
14         self.setWindowTitle("My App")
15
16         widget = QLabel("Hello")
17         widget.setPixmap(QPixmap(os.path.join(basedir, "otje.jpg")))
18
19         self.setCentralWidget(widget)
20
21 app = QApplication(sys.argv)
22
23 window = MainWindow()
24 window.show()
25
26 app.exec()
```

① Directorio de trabajo actual.

② Ruta base (relativa al archivo del script).



No te preocupes si esta parte aún no la entiendes del todo; más adelante la veremos con más profundidad.

### Recuerda:

- `os.getcwd()` indica dónde estás ejecutando el programa.
- `os.path.dirname(__file__)` indica dónde está guardado el script.

Para cargar imágenes, siempre es más fiable usar la segunda.

## Escalar la imagen

Por defecto, la imagen mantiene su tamaño y proporción original.

Si quieres que **ocupe todo el espacio disponible** dentro de la ventana, puedes activar el modo de escalado:

```
1 widget.setPixmap(QPixmap(os.path.join(basedir, "otje.jpg")))
2 widget.setScaledContents(True)
```

► **Ejecuta el programa.** Redimensiona la ventana y verás cómo la imagen se deforma para ajustarse.



Mostrando un pixmap con QLabel en Windows, macOS y Ubuntu.

## ► Ejercicio propuesto 4

### QCheckBox

El widget **QCheckBox** permite al usuario activar o desactivar una opción concreta mediante una **casilla de verificación**.

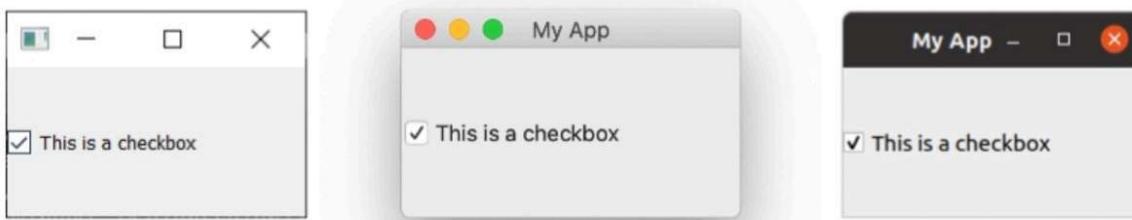
Se usa cuando una elección es independiente de las demás, a diferencia de los **QRadioButton**, que representan opciones excluyentes.

Un ejemplo típico sería permitir al usuario seleccionar varias preferencias: “Activar notificaciones”, “Modo oscuro”, “Guardar sesión automáticamente”.

Como con otros widgets de Qt, existen distintas opciones configurables para modificar su comportamiento.

```
1 import sys
2 from PySide6.QtCore import Qt
3 from PySide6.QtWidgets import QApplication, QCheckBox, QMainWindow
4
5 class MainWindow(QMainWindow):
6     def __init__(self):
7         super().__init__()
8
9         self.setWindowTitle("My App")
10
11        widget = QCheckBox("This is a checkbox")
12        widget.setCheckState(Qt.Checked)
13
14        # Para estado tri: widget.setCheckState(Qt.PartiallyChecked)
15        # 0: widget.setTristate(True)
16
17        widget.stateChanged.connect(self.show_state)
18
19        self.setCentralWidget(widget)
20
21    def show_state(self, s):
22        state = Qt.CheckState(s)
23        print("¿Checked?:", state == Qt.CheckState.Checked)
24        print("Estado bruto:", s, "| Enum:", state)
25
26 app = QApplication(sys.argv)
27
28 window = MainWindow()
29 window.show()
30
31 app.exec()
```

► Ejecuta el programa. Verás una casilla con texto al lado.



*QCheckBox en Windows, macOS y Ubuntu.*

Puedes configurar el estado de una casilla desde el código usando `.setChecked()` o `.setCheckState()`:

Flag	Comportamiento
<code>Qt.Checked</code>	La casilla está marcada
<code>Qt.Unchecked</code>	La casilla está desmarcada
<code>Qt.PartiallyChecked</code>	La casilla está parcialmente marcada

Una casilla en estado `Qt.PartiallyChecked` se denomina *tri-state* porque no está ni activada ni desactivada completamente.

Se representa visualmente con un recuadro atenuado o sombreado, y es útil en **estructuras jerárquicas**.

Por ejemplo, una casilla “Seleccionar todo” podría aparecer parcialmente marcada si solo algunas subopciones están activas.

Puedes activar el modo tri-state de dos formas:

```

1 widget.setCheckState(Qt.PartiallyChecked)
2 # o
3 widget.setTristate(True)

```

💡 Durante la ejecución, Qt muestra el estado interno como un entero del enumerado `Qt.CheckState`:

- `Checked = 2`
- `Unchecked = 0`
- `PartiallyChecked = 1`

No es necesario memorizar estos valores; basta con comparar así:

```

1 if state == Qt.Checked:
2     print("Marcado")

```

🌐 Relación entre estados y valores lógicos

Estado	Valor lógico	CheckState (entero)
Qt.Checked	True	2
Qt.Unchecked	False	0
Qt.PartiallyChecked	None	1

Esto ayuda a entender cómo se traducen los estados en condiciones o estructuras de control.

## Ejemplo 2 – Casilla con estado inicial

```

1 import sys
2 from PySide6.QtCore import Qt
3 from PySide6.QtWidgets import QApplication, QCheckBox, QMainWindow
4
5 class MainWindow(QMainWindow):
6     def __init__(self):
7         super().__init__()
8         self.setWindowTitle("Casilla con estado inicial")
9
10        widget = QCheckBox("Modo oscuro activado por defecto")
11        widget.setChecked(True)
12        widget.stateChanged.connect(self.mostrar_estado)
13
14        self.setCentralWidget(widget)
15
16    def mostrar_estado(self, s):
17        print("Estado:", s == Qt.Checked)
18
19 app = QApplication(sys.argv)
20 window = MainWindow()
21 window.show()
22 app.exec()
```

 **Qué muestra:** cómo definir el estado inicial de una casilla (`setChecked(True)`).

### Ejemplo 3 – Casilla tri-state

```
1 import sys
2 from PySide6.QtCore import Qt
3 from PySide6.QtWidgets import QApplication, QCheckBox, QMainWindow
4
5 class MainWindow(QMainWindow):
6     def __init__(self):
7         super().__init__()
8         self.setWindowTitle("Casilla tri-state")
9
10        widget = QCheckBox("Seleccionar todo")
11        widget.setTristate(True)
12        widget.setCheckState(Qt.PartiallyChecked)
13        widget.stateChanged.connect(self.mostrar_estado)
14
15        self.setCentralWidget(widget)
16
17    def mostrar_estado(self, s):
18        if s == Qt.CheckState.Checked.value:
19            print("Marcado completamente")
20        elif s == Qt.CheckState.Unchecked.value:
21            print("Desmarcado")
22        else:
23            print("Marcado parcialmente")
24
25 app = QApplication(sys.argv)
26 window = MainWindow()
27 window.show()
28 app.exec()
```

➤ **Qué muestra:** cómo usar el modo *tri-state* (`Qt.PartiallyChecked`) y detectar los tres posibles estados.

### ► Ejercicio propuesto 5

## QComboBox

**QComboBox** es un **desplegable** (drop-down list) que permite al usuario **elegir una sola opción** de entre una lista.

Cuando está cerrado, muestra el elemento seleccionado junto a una flecha para desplegar la lista completa.

Se usa en muchos programas para elegir cosas como fuentes, tamaños, idiomas o categorías.

 En Qt existe incluso una versión especializada para fuentes: **QFontComboBox**.

### Crear y llenar un QComboBox

Los elementos se pueden añadir con el método **.addItems()**, que recibe una lista de textos.

Los elementos se muestran en el mismo orden en que se añaden.

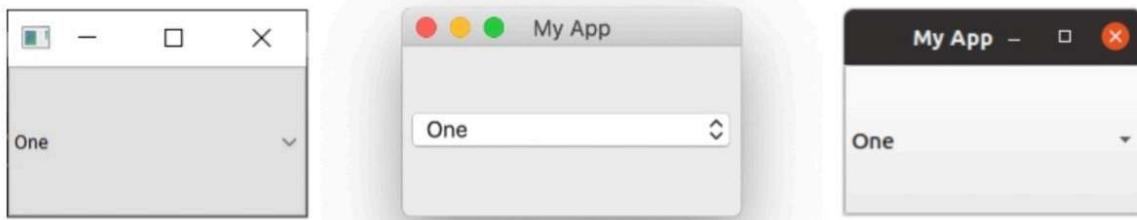
```
1 widget = QComboBox()  
2 widget.addItems( [ "Uno", "Dos", "Tres" ] )
```

### Ejemplo

```

1 import sys
2 from PySide6.QtWidgets import QApplication, QMainWindow, QComboBox
3
4 class MainWindow(QMainWindow):
5     def __init__(self):
6         super().__init__()
7         self.setWindowTitle("Ejemplo QComboBox")
8
9         combo = QComboBox()
10        combo.addItems(["Uno", "Dos", "Tres"])
11
12        combo.currentIndexChanged.connect(self.cambio_indice)
13        combo.currentTextChanged.connect(self.cambio_texto)
14
15        self.setCentralWidget(combo)
16
17    def cambio_indice(self, i):
18        print("Índice seleccionado:", i)
19
20    def cambio_texto(self, s):
21        print("Texto seleccionado:", s)
22
23
24 app = QApplication(sys.argv)
25 window = MainWindow()
26 window.show()
27 app.exec()

```



*QComboBox en Windows, macOS y Ubuntu.*

### ✳️ Qué ocurre:

1. Aparece un desplegable con tres opciones.
2. Cada vez que cambias la selección:
  - `currentIndexChanged` envía el **número de índice** (0, 1, 2...).
  - `currentTextChanged` envía el **texto** de la opción elegida.

## Señales más comunes

Señal	Qué envía	Cuándo se activa
currentIndexChanged d	índice (int)	Al cambiar la selección
currentTextChanged	texto (str)	Al cambiar el texto mostrado

💡 En la práctica, casi siempre nos interesa más el **texto** que el índice.

## Combobox editable

También puedes permitir que el usuario **escriba su propia opción**.

Para eso, activa el modo editable:

```
1 combo.setEditable(True)
```

Si haces esto, el usuario podrá escribir un nuevo valor y pulsar **Enter** para añadirlo.

## Políticas de inserción ( `InsertPolicy` )

Cuando el usuario escribe un valor nuevo, Qt necesita saber **dónde colocarlo**.

Esto se controla con una *política de inserción*:

<b>Flag</b>	<b>Comportamiento</b>
<code>QComboBox.NoInsert</code>	No se añade el texto nuevo
<code>QComboBox.InsertAtTop</code>	Se añade al principio
<code>QComboBox.InsertAtBottom</code>	Se añade al final
<code>QComboBox.InsertAtCurrent</code>	Sustituye el elemento actual
<code>QComboBox.InsertAfterCurrent</code>	Se añade justo después
<code>QComboBox.InsertBeforeCurrent</code>	Se añade justo antes
<code>QComboBox.InsertAlphabetically</code>	Se inserta en orden alfabético

Para aplicarla:

```
1 combo.setInsertPolicy(QComboBox.InsertAlphabetically)
```

También puedes limitar el número total de elementos:

```
1 combo.setMaxCount(5)
```

Ejemplo completo

```
1 import sys
2 from PySide6.QtWidgets import QApplication, QMainWindow, QComboBox
3
4 class MainWindow(QMainWindow):
5     def __init__(self):
6         super().__init__()
7         self.setWindowTitle("QComboBox editable")
8
9         combo = QComboBox()
10        combo.addItems(["Uno", "Dos", "Tres"])
11        combo.setEditable(True)
12        combo.setInsertPolicy(QComboBox.InsertAlphabetically)
13        combo.setMaxCount(5)
14
15        combo.currentIndexChanged.connect(self.cambio_indice)
16        combo.currentTextChanged.connect(self.cambio_texto)
17
18        self.setCentralWidget(combo)
19
20    def cambio_indice(self, i):
21        print("Índice:", i)
22
23    def cambio_texto(self, s):
24        print("Texto:", s)
25
26
27 app = QApplication(sys.argv)
28 window = MainWindow()
29 window.show()
30 app.exec()
```

## Prueba tú

1. Ejecuta el programa.
2. Escribe un valor nuevo y pulsa **Enter**.
3. Observa cómo se inserta **en orden alfabético**.
4. Intenta añadir más de 5 elementos: verás que no deja.

## ► Ejercicio propuesto 6

### QLineEdit

El widget **QLineEdit** es una **caja de texto de una sola línea**, pensada para que el usuario escriba información breve: un nombre, una dirección de correo, una contraseña o un número.

Se usa cuando **no hay una lista cerrada de opciones**, sino que la persona usuaria debe introducir libremente su propio texto.

### Crear un campo de texto

```
1 from PySide6.QtWidgets import QLineEdit  
2  
3 campo = QLineEdit()
```

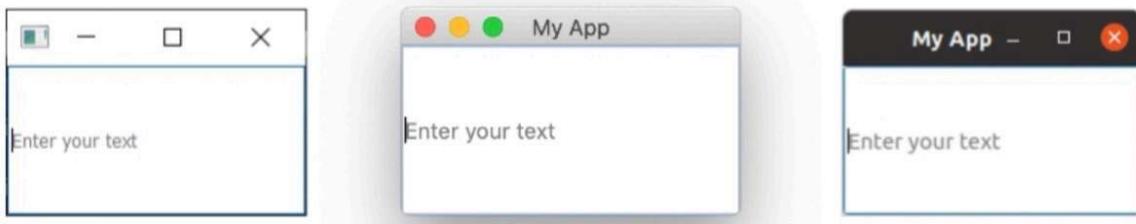
Una vez creado, puedes configurar su comportamiento con métodos como:

```
1 campo.setMaxLength(20) # Número máximo de caracte  
2 campo.setPlaceholderText("Escribe aquí") # Texto de ayuda inicial  
3 campo.setReadOnly(True) # Solo lectura (opcional)
```

### Ejemplo

```
1 import sys
2 from PySide6.QtWidgets import QApplication, QMainWindow, QLineEdit
3
4 class MainWindow(QMainWindow):
5     def __init__(self):
6         super().__init__()
7         self.setWindowTitle("Ejemplo QLineEdit")
8
9         texto = QLineEdit()
10        texto.setMaxLength(10)
11        texto.setPlaceholderText("Introduce tu nombre")
12
13        texto.returnPressed.connect(self.mostrar_mensaje)
14        texto.textChanged.connect(self.texto_modificado)
15        texto.textEdited.connect(self.texto_editado)
16
17        self.setCentralWidget(texto)
18        self.texto = texto
19
20    def mostrar_mensaje(self):
21        print("Se pulsó Enter")
22        self.texto.setText("¡Hola!")
23
24    def texto_modificado(self, s):
25        print("Texto modificado:", s)
26
27    def texto_editado(self, s):
28        print("Texto editado por el usuario:", s)
29
30 app = QApplication(sys.argv)
31 window = MainWindow()
32 window.show()
33 app.exec()
```

Qué hace este ejemplo



*QLineEdit en Windows, macOS y Ubuntu.*

1. Muestra una caja de texto con una pista visual ("Introduce tu nombre").
2. Si el usuario escribe y pulsa **Enter**, el contenido cambia a "¡Hola!".
3. Cada cambio de texto se muestra por consola.

## Señales

Señal	Cuándo se activa	Emite
<code>returnPressed</code>	Al pulsar Enter	(sin parámetros)
<code>selectionChanged</code>	Al cambiar la selección del texto	(sin parámetros)
<code>textChanged</code>	Cada vez que cambia el texto (usuario o código)	<code>str</code>
<code>textEdited</code>	Solo si el usuario edita manualmente el texto	<code>str</code>

👉 En resumen:

- Usa `textEdited` si te interesa solo lo que el usuario escribe.
- Usa `textChanged` si también quieras detectar cambios hechos por el programa.

## Métodos útiles

Método	Qué hace
<code>.setMaxLength(n)</code>	Limita el número de caracteres.
<code>.setPlaceholderText("texto")</code>	Muestra texto guía cuando está vacío.
<code>.setReadOnly(True)</code>	Hace el campo no editable.
<code>.clear()</code>	Borra el contenido.
<code>.text()</code>	Devuelve el texto actual.

### Máscaras de entrada (Input Mask)

Una *máscara de entrada* define qué caracteres puede introducir el usuario y en qué formato.

Por ejemplo, para forzar el formato de una dirección IP:

```
1 campo.setInputMask("000.000.000.000;_")
```

→ Solo permitirá números y puntos en las posiciones correctas (por ejemplo: 192.168.001.010).

Algunos ejemplos adicionales:

Máscara	Formato permitido	Ejemplo válido
"00000"	Solo 5 dígitos	41013
5 letras mayúsculas	SEVIL	
"(000) 000-000"	Número de teléfono	(954) 123-456

### Ejemplo con máscara

```
1 import sys
2 from PySide6.QtWidgets import QApplication, QMainWindow, QLineEdit
3
4 class MainWindow(QMainWindow):
5     def __init__(self):
6         super().__init__()
7         self.setWindowTitle("Máscara de entrada")
8
9         ip = QLineEdit()
10        ip.setInputMask("000.000.000.000;_")
11        ip.setPlaceholderText("Introduce una dirección IP")
12
13        self.setCentralWidget(ip)
14
15 app = QApplication(sys.argv)
16 window = MainWindow()
17 window.show()
18 app.exec()
```

## ► Ejercicio propuesto 7

### QTextEdit

**QTextEdit** es un **campo de texto multilínea** que permite escribir, mostrar y editar **párrafos completos**.

A diferencia de **QLineEdit**, que se usa para textos cortos (una sola línea), **QTextEdit** sirve para:

- Comentarios, descripciones o mensajes largos.
- Mostrar contenido de texto formateado (por ejemplo, HTML o Markdown).
- Crear editores de texto sencillos.

### Crear un QTextEdit

```
1 from PySide6.QtWidgets import QTextEdit  
2  
3 texto = QTextEdit()
```

Por defecto, se abre un cuadro vacío en el que se puede escribir libremente.

Puedes personalizar su contenido inicial y comportamiento:

```
1 texto.setPlainText("Escribe aquí tu texto...")  
2 texto.setPlaceholderText("Introduce varias líneas de texto")
```

## Ejemplo

```
1 import sys  
2 from PySide6.QtWidgets import QApplication, QMainWindow, QTextEdit  
3  
4 class MainWindow(QMainWindow):  
5     def __init__(self):  
6         super().__init__()  
7         self.setWindowTitle("Ejemplo QTextEdit")  
8  
9         campo = QTextEdit()  
10        campo.setPlaceholderText("Escribe aquí tu comentario...")  
11  
12        # Conectamos señales  
13        campo.textChanged.connect(self.texto_modificado)  
14  
15        self.setCentralWidget(campo)  
16        self.campo = campo  
17  
18    def texto_modificado(self):  
19        print("El texto ha cambiado")  
20        print(self.campo.toPlainText())  
21  
22    app = QApplication(sys.argv)  
23    window = MainWindow()  
24    window.show()  
25    app.exec()
```

### Qué ocurre

1. El programa muestra un cuadro de texto multilínea.
2. Cada vez que el usuario escribe o borra algo, la señal `textChanged` se lanza.
3. Se imprime en consola el texto completo usando `toPlainText()`.

### Principales métodos

Método	Descripción
<code>.setPlainText("texto")</code>	Inserta texto plano sin formato.
<code>.toPlainText()</code>	Devuelve el texto como cadena normal.
<code>.setHtml("&lt;b&gt;Texto&lt;/b&gt;")</code>	Inserta texto con formato HTML.
<code>.toHtml()</code>	Devuelve el contenido en formato HTML.
<code>.setReadOnly(True)</code>	Impide la edición (solo lectura).
<code>.clear()</code>	Borra todo el contenido.
<code>.append("texto")</code>	Añade texto al final, con salto de línea.
<code>.setPlaceholderText("texto")</code>	Muestra texto guía cuando está vacío.

### Ejemplo con texto inicial y solo lectura

```
1 import sys
2 from PySide6.QtWidgets import QApplication, QMainWindow, QTextEdit
3
4 class MainWindow(QMainWindow):
5     def __init__(self):
6         super().__init__()
7         self.setWindowTitle("QTextEdit solo lectura")
8
9         texto = QTextEdit()
10        texto.setPlainText("Bienvenido al módulo de Desarrollo de Software")
11        texto.setReadOnly(True)
12
13        self.setCentralWidget(texto)
14
15 app = QApplication(sys.argv)
16 window = MainWindow()
17 window.show()
18 app.exec()
```

Este ejemplo muestra un campo de texto **no editable**, ideal para mostrar instrucciones o resultados.

### Añadir texto dinámicamente

Puedes añadir texto con `.append()`, muy útil para mostrar mensajes o registros.

```
1 self.campo.append("Nuevo mensaje añadido.")
```

Esto agrega el texto al final, sin borrar lo anterior.

### Señales

Señal	Cuándo se emite	Qué hace
<code>textChanged</code>	Cada vez que cambia el contenido	Detecta modificaciones (usuario o código)
<code>cursorPositionChanged</code>	Cuando el cursor cambia de línea o posición	Permite conocer la ubicación del cursor

💡 `QTextEdit` no tiene `returnPressed` como `QLineEdit`, porque admite saltos de línea.

### Ejemplo con HTML

`QTextEdit` también puede mostrar texto con formato básico usando HTML:

```
1 texto = QTextEdit()
2 texto.setHtml("<h2>Título</h2><p>Esto es un <b>texto en negrita</b></p>")
```

También puedes recuperar el contenido con `.toHtml()` para guardarlo o procesarlo.

### En resumen

Característica	<code>QLineEdit</code>	<code>QTextEdit</code>
Líneas de texto	Una sola	Múltiples
Formato HTML	✗ No	✓ Sí
Señal Enter	<code>returnPressed</code>	No disponible
Texto plano	<code>.text()</code>	<code>.toPlainText()</code>
Texto formateado	—	<code>.setHtml()</code> / <code>.toHtml()</code>
Uso típico	Formularios, campos cortos	Comentarios, notas, textos largos

## ► Ejercicio propuesto 8

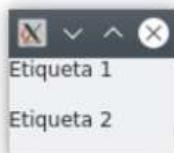
### 3. Contenedores de componentes. Diseño.

En Qt podemos asignar tamaños máximos, mínimos y fijos a los componentes, desplazarlos, etc. Pero no resulta cómodo tener que estar realizando cálculos para que nuestra interfaz resulte amigable. Además, sería muy difícil mantener el código generado si el número de componentes aumenta.

Por ello, en este apartado se estudia cómo gestionar todo esto de forma más cómoda mediante **layouts**.

#### 3.1. Layouts

Hasta ahora hemos visto ventanas con un único componente y componentes contenidos dentro de otros componentes, como es el caso del ejemplo siguiente. En la imagen puedes ver el resultado.



```
1  from PySide6.QtWidgets import QApplication, QLabel, QWidget
2  class Ventana(QWidget):
3      def __init__(self):
4          QWidget.__init__(self)
5          self.setWindowTitle("Ventana")
6          # Creamos dos QLabel's con el componente como parent
7          self.label1 = QLabel("Etiqueta 1", self)
8          self.label2 = QLabel("Etiqueta 2", self)
9          # Necesitamos mover la segunda 30 píxeles hacia abajo para
10         # que no se solape con la primera
11         self.label2.move(0, 30)
12
13 if __name__ == "__main__":
14     app = QApplication([])
15     ventana = Ventana()
16     # Mostramos la ventana
17     ventana.show()
18     app.exec()
```

Pero ¿qué pasa si queremos añadir más componentes, tanto horizontalmente como verticalmente? ¿Qué pasa si redimensionamos la ventana? Tendríamos que ir calculando el número de píxeles a desplazar y el espacio que ocupan en la interfaz no quedaría modificada. Por eso, en este apartado vamos a estudiar una forma más eficiente de gestionar todo esto a través de layouts: disposiciones que podemos aplicar a una interfaz para ordenar sus componentes. Con la combinación de estos layouts es posible definir el diseño de cualquier interfaz gráfica de usuario.

### Tipos de layouts en Qt

Antes de profundizar en cada tipo de layout, conviene tener una visión general de las opciones disponibles. Cada disposición organiza los componentes de una forma distinta, y elegir la adecuada dependerá del tipo de interfaz que queramos construir.

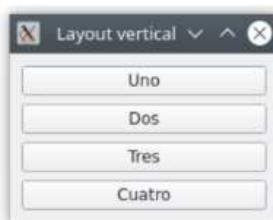
Layout	Organización	Ejemplo de uso
<b>QVBoxLayout</b>	Coloca los componentes en vertical, uno debajo de otro.	Listas de botones, formularios simples.
<b>QHBoxLayout</b>	Coloca los componentes en horizontal, uno junto a otro.	Barras de herramientas, menús o paneles laterales.
<b>QGridLayout</b>	Distribuye los componentes en una cuadrícula de filas y columnas.	Formularios o interfaces con alineación precisa.
<b>QFormLayout</b>	Diseñado para parejas de etiqueta y campo de entrada.	Formularios de datos o configuraciones.
<b>QStackedLayout</b>	Apila varios componentes en el mismo espacio, mostrando solo uno a la vez.	Pestañas, menús o pantallas que cambian según la acción del usuario.

 **Consejo:** en la mayoría de los casos, combinar layouts (por ejemplo, un `QVBoxLayout` que contiene un `QHBoxLayout` dentro) permite crear diseños mucho más flexibles y adaptables.

### 3.2. QVBoxLayout

El **QVBoxLayout** organiza los componentes en **vertical**, colocándolos uno debajo de otro como si fueran cajas apiladas.

Este tipo de disposición resulta muy útil cuando queremos que los elementos se repartan de forma equilibrada a lo largo de una columna.



Si redimensionamos la ventana, los botones se adaptan automáticamente al ancho y se reparten de forma proporcional en vertical.

No es necesario calcular posiciones ni tamaños manualmente.

### ¿A qué widgets se les puede aplicar un layout?

Un **layout** se puede asignar a cualquier componente que **herede de QWidget**, ya que este tipo de objetos puede actuar como contenedor de otros.

Sin embargo, hay **algunos widgets especiales** (como `QMainWindow`, `QScrollArea`, `QSplitter` o `QTabWidget`) que **no permiten asignarles un layout directamente** porque ya gestionan su propio sistema interno de áreas.

Por eso, cuando trabajamos con `QMainWindow`, es necesario crear un **widget contenedor** (por ejemplo, un `QWidget` normal), aplicarle el layout y después establecerlo como contenido central mediante `setCentralWidget()`.

```
1  from PySide6.QtWidgets import (
2      QApplication, QMainWindow, QWidget, QVBoxLayout, QPushButton
3  )
4
5  class VentanaPrincipal(QMainWindow):
6      def __init__(self):
7          super().__init__()
8          self.setWindowTitle("Layout vertical")
9
10     # 1 Creamos el layout vertical
11     layout_vertical = QVBoxLayout()
12
13     # 2 Creamos el contenedor principal y le asignamos el layout
14     componente_principal = QWidget()
15     componente_principal.setLayout(layout_vertical)
16
17     # 3 Lo establecemos como contenido central del QMainWindow
18     self.setCentralWidget(componente_principal)
19
20     # 4 Añadimos los widgets al layout (después de asignarlos)
21     layout_vertical.addWidget(QPushButton("Uno"))
22     layout_vertical.addWidget(QPushButton("Dos"))
23     layout_vertical.addWidget(QPushButton("Tres"))
24     layout_vertical.addWidget(QPushButton("Cuatro"))
25
26
27 app = QApplication([])
28 ventana = VentanaPrincipal()
29 ventana.show()
30 app.exec()
```

Hemos definido un componente principal de tipo QWidget al que le asignamos un layout vertical. A este layout le añadimos los componentes que vamos a utilizar. Si ahora probamos a redimensionar la ventana, los componentes cambian automáticamente de tamaño para ajustarse al ancho de la ventana y repartirse de forma equitativa verticalmente.



### Nota

Si intentáramos aplicar el layout directamente al `QMainWindow` con `self.setLayout(layout_vertical)`, el programa mostraría un error, ya que `QMainWindow` **no acepta layouts directamente**.

Por eso, el uso del `QWidget` intermedio es necesario y una buena práctica habitual en PySide6.

### Ejercicio de clase

Prueba a añadir un quinto botón al layout.

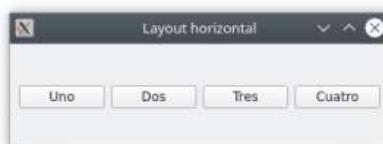
¿Qué ocurre cuando cambias el tamaño de la ventana?

¿Se adaptan todos los botones por igual?

### 3.3. QHBoxLayout

El **QHBoxLayout** organiza los componentes en **horizontal**, colocándolos uno al lado del otro en una misma fila.

Es ideal para crear barras de herramientas, menús o zonas donde los elementos deban alinearse de izquierda a derecha.



Si redimensionamos la ventana, los botones **se expanden horizontalmente**, manteniendo la proporción entre ellos.

El alto permanece constante, pero el ancho se ajusta al espacio disponible.

```
1  from PySide6.QtWidgets import (
2      QApplication, QMainWindow, QWidget, QHBoxLayout, QPushButton
3 )
4  class VentanaPrincipal(QMainWindow):
5      def __init__(self):
6          super().__init__()
7          self.setWindowTitle("Layout horizontal")
8          # Creamos un objeto layout horizontal
9          layout_horizontal = QHBoxLayout()
10         # Creamos un componente principal para la ventana
11         componente_principal = QWidget()
12         # Le asignamos el layout vertical como disposición
13         componente_principal.setLayout(layout_horizontal)
14         self.setCentralWidget(componente_principal)
15         # Añadimos cuatro botones al layout vertical
16         layout_horizontal.addWidget(QPushButton("Uno"))
17         layout_horizontal.addWidget(QPushButton("Dos"))
18         layout_horizontal.addWidget(QPushButton("Tres"))
19         layout_horizontal.addWidget(QPushButton("Cuatro"))
20
21     app = QApplication([])
22     ventana = VentanaPrincipal()
23     ventana.show()
24     app.exec()
```

## Observa

Cuando redimensionas la ventana:

- Los botones **crecen o se reducen horizontalmente**, manteniendo su alineación.
- El espacio entre ellos se adapta automáticamente.
- El alto de los botones no cambia, solo su ancho.



## Ejercicio de clase

Cambia los textos de los botones por nombres más largos y redimensiona la ventana.

¿Notas cómo el layout redistribuye el espacio para mantenerlos en una sola fila?

Prueba también a añadir un quinto botón y observa el comportamiento.

### 3.4. QGridLayout

El **QGridLayout** organiza los componentes en forma de **cuadrícula**, con filas y columnas.

Funciona de manera similar a una tabla: cada celda puede contener un widget, y puedes decidir cuántas filas y columnas ocupará cada uno.

Este tipo de layout es muy útil cuando quieras alinear elementos con precisión, como formularios o paneles de control.



Cada posición se indica con su fila y columna, igual que en una hoja de cálculo.

Algunos widgets pueden ocupar más de una celda horizontal o verticalmente.

```
1  from PySide6.QtWidgets import (
2      QApplication, QMainWindow, QWidget, QGridLayout, QPushButton
3 )
4  class VentanaPrincipal(QMainWindow):
5      def __init__(self):
6          super().__init__()
7          self.setWindowTitle("Layout cuadrícula")
8          # Creamos un objeto layout cuadrícula
9          layout_cuadricula = QGridLayout()
10         componente_principal = QWidget()
11         componente_principal.setLayout(layout_cuadricula)
12         self.setCentralWidget(componente_principal)
13         # Añadimos cuatro botones a la primera fila
14         # Los números indican la fila y la columna donde situar
15         layout_cuadricula.addWidget(QPushButton("0,0"), 0, 0)
16         layout_cuadricula.addWidget(QPushButton("0,1"), 0, 1)
17         layout_cuadricula.addWidget(QPushButton("0,2"), 0, 2)
18         layout_cuadricula.addWidget(QPushButton("0,3"), 0, 3)
19         # Añadimos un botón a la segunda fila que ocupe cuatro columnas
20         layout_cuadricula.addWidget(QPushButton("1,0-3"), 1, 0, 1, 3)
21         # Añadimos dos botones a la tercera fila, que ocupen dos columnas
22         layout_cuadricula.addWidget(QPushButton("2,0-1"), 2, 0, 1, 1)
23         layout_cuadricula.addWidget(QPushButton("2,2-3"), 2, 2, 1, 2)
24
25     app = QApplication([])
26     ventana = VentanaPrincipal()
27     ventana.show()
28     app.exec()
```

## Observa

- Cada botón ocupa una posición específica en la cuadrícula.
- Algunos botones se extienden por varias columnas o filas.
- Si redimensionas la ventana, los componentes se ajustan automáticamente, manteniendo la estructura de la cuadrícula.

## Ejercicio de clase

Añade un nuevo botón en la posición (3, 1) y haz que ocupe dos columnas.

Observa cómo se reorganiza el diseño y cómo se mantiene alineado con los demás elementos.

Luego, cambia el número de columnas que ocupa el botón central y analiza cómo afecta al resto del layout.

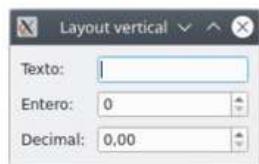
### 3.5. QFormLayout

El **QFormLayout** está pensado para crear **formularios**: una disposición en la que cada fila contiene una **etiqueta** y un **campo de entrada**.

Es una forma muy cómoda de construir pantallas de registro, configuración o edición de datos.

Este layout también se puede usar para mostrar información (deshabilitando los campos de entrada).

Un formulario básico con tres campos podría representarse así:



Cada fila tiene una etiqueta a la izquierda y un componente de entrada a la derecha.

```
1  from PySide6.QtWidgets import (
2      QApplication, QMainWindow, QWidget, QFormLayout,
3      QLabel, QLineEdit, QSpinBox, QDoubleSpinBox
4  )
5  class VentanaPrincipal(QMainWindow):
6      def __init__(self):
7          super().__init__()
8          self.setWindowTitle("Layout vertical")
9          # Creamos un objeto layout formulario
10         layout_formulario = QFormLayout()
11         componente_principal = QWidget()
12         componente_principal.setLayout(layout_formulario)
13         self.setCentralWidget(componente_principal)
14         # Cada fila contendrá una etiqueta y un componente de entrada
15         layout_formulario.addRow(QLabel("Texto:"), QLineEdit())
16         layout_formulario.addRow(QLabel("Entero:"), QSpinBox())
17         layout_formulario.addRow(QLabel("Decimal:"), QDoubleSpinBox())
18
19 app = QApplication([])
20 ventana = VentanaPrincipal()
21 ventana.show()
22 app.exec()
```

## Observa

- Cada fila se compone de dos partes: la **etiqueta** y el **campo de entrada**.
- El layout ajusta automáticamente los tamaños y mantiene las etiquetas alineadas.
- Si cambias el tamaño de la ventana, todos los campos se adaptan proporcionalmente.

## Ejercicio de clase

Añade una nueva fila al formulario con una lista desplegable (**QComboBox**) que permita elegir una provincia.

Luego, desactiva uno de los campos con `.setEnabled(False)` y observa cómo sigue manteniendo la alineación.

Prueba a ampliar la ventana para ver cómo se reajustan los campos.

### 🌱 ¿Sabías que...?

El **QFormLayout** permite incluir también una sola columna de componentes si se añaden las filas con un único argumento:

```
layout_formulario.addRow(QLabel("Título principal"))
```

Además, puedes combinarlo dentro de un **QVBoxLayout** o un **QGridLayout** para incluir formularios dentro de interfaces más grandes, manteniendo una estructura clara y limpia.

## 3.6. QStackedLayout

El **QStackedLayout** permite **apilar varios componentes en el mismo espacio**, mostrando solo uno a la vez.

Imagina una pila de cartas en la que solo se ve la que está arriba: las demás siguen ahí, pero ocultas.

Este layout es muy útil cuando quieras cambiar el contenido mostrado sin cambiar de ventana, como ocurre con las **pestañas** o los **menús de navegación**.

Para gestionar qué elemento es visible utilizamos `setCurrentIndex` o `setCurrentWidget`.

```
1  from PySide6.QtWidgets import (
2      QApplication, QMainWindow, QWidget, QPushButton,
3      QStackedLayout, QLabel, QVBoxLayout, QHBoxLayout
4  )
5  class VentanaPrincipal(QMainWindow):
6      def __init__(self):
7          super().__init__()
8          self.setWindowTitle("Layout apilado")
9          layout_principal = QHBoxLayout()
10         componente_principal = QWidget()
11         componente_principal.setLayout(layout_principal)
12         self.setCentralWidget(componente_principal)
13
14         # Creamos un QStackedLayout y añadimos cuatro "capas"
15         self.pila = QStackedLayout()
16         self.pila.addWidget(QLabel("Capa 1"))
17         self.pila.addWidget(QLabel("Capa 2"))
18         self.pila.addWidget(QLabel("Capa 3"))
19
20         # Creamos un layout vertical con tres botones
21         # Cada botón hará visible una capa a través de la ranura
22         layout_botones = QVBoxLayout()
23         boton1 = QPushButton("Ver capa 1")
24         boton1.clicked.connect(self.activar_capa1)
25         boton2 = QPushButton("Ver capa 2")
26         boton2.clicked.connect(self.activar_capa2)
27         boton3 = QPushButton("Ver capa 3")
28         boton3.clicked.connect(self.activar_capa3)
29         layout_botones.addWidget(boton1)
30         layout_botones.addWidget(boton2)
31         layout_botones.addWidget(boton3)
32
33         # Añadimos los layouts al layout principal
34         layout_principal.addLayout(self.pila)
35         layout_principal.addLayout(layout_botones)
36
37     def activar_capa1(self):
38         self.pila.setCurrentIndex(0)
39     def activar_capa2(self):
40         self.pila.setCurrentIndex(1)
41     def activar_capa3(self):
42         self.pila.setCurrentIndex(2)
43
```

```

44 app = QApplication([])
45 ventana = VentanaPrincipal()
46 ventana.show()
47 app.exec()

```

El resultado es una interfaz parecida al uso de pestañas, pero con botones.



### Observa

- Solo una capa del layout está visible a la vez.
- Los botones permiten cambiar la capa activa dinámicamente.
- Todas las capas comparten el mismo espacio dentro de la ventana, por lo que no es necesario crear nuevas ventanas o pestañas.

### Cómo funciona `addLayout()`

El método `addLayout()` permite **incluir un layout dentro de otro**.

Su sintaxis básica es:

```
layout_padre.addLayout(layout_hijo)
```

Esto indica que el **layout hijo** se colocará como un bloque dentro del **layout padre**.

El layout hijo puede contener sus propios widgets (botones, etiquetas, formularios, etc.), y conservará su estructura interna dentro del layout principal.

### Ejercicio de clase

Crea una cuarta capa con un `QLabel` que muestre el texto "Bienvenido a la capa 4".

Añade un botón extra que la active.

Luego, modifica uno de los botones para que use `setCurrentWidget()` en lugar de `setCurrentIndex()` y observa que el resultado es el mismo.

### ¿Sabías que...?

El `QStackedLayout` se usa internamente en widgets más complejos como `QStackedWidget` o `QTabWidget`.

Estos añaden pestañas o animaciones por defecto, pero la lógica base es la misma: **una pila de componentes donde solo uno es visible**.

### ► Ejercicio propuesto 9

## 4. Barras de herramientas, barra de estado y menús

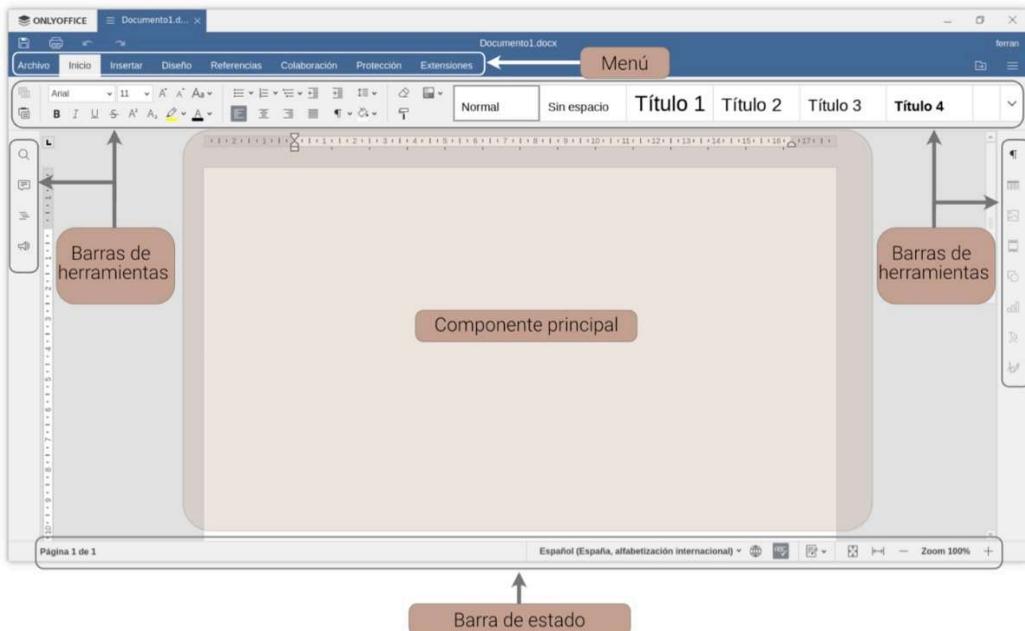
Hasta ahora hemos trabajado con componentes y conjuntos de ellos, pero una aplicación también está compuesta por otros elementos como los menús, las barras de herramientas o la barra de estado.

En este apartado veremos cómo añadir estos elementos a nuestras aplicaciones usando **Qt**.

### 4.1. Estructura general de una aplicación

Si observas la ventana principal de la mayoría de las aplicaciones de escritorio, su estructura suele seguir un patrón similar:

- **Menú:** normalmente en forma de lista desplegable, aunque a veces se presentan como pestañas.
- **Barras de herramientas:** agrupan acciones habituales a las que se accede con un solo clic.
- **Componente principal:** ocupa la parte central y contiene la funcionalidad principal.
- **Barra de estado:** muestra información sobre el estado actual o alguna configuración activa.



En Qt, la clase que nos permite integrar todos estos elementos en una misma ventana es **QMainWindow**.

Elemento	Descripción	Clase principal
Menú	Agrupa acciones organizadas por categorías (Archivo, Edición, etc.)	<b>QMenuBar</b>
Barra de herramientas	Permite ejecutar acciones comunes de forma rápida mediante iconos	<b>QToolBar</b>
Barra de estado	Muestra mensajes, estados o información contextual	<b>QStatusBar</b>

## 4.2. QActions

Antes de empezar con los menús o las barras, debemos conocer un componente clave: **QAction**.

En una aplicación, una misma funcionalidad puede activarse desde distintos lugares: un menú, un botón de la barra de herramientas o un atajo de teclado. Las **acciones** (**QAction**) permiten centralizar esa funcionalidad para reutilizarla en distintos elementos de la interfaz.

Por ejemplo, en un procesador de textos, al pulsar:

- Menú → *Archivo > Guardar*
- Botón de la barra de herramientas *Guardar*
- Atajo de teclado ***Ctrl + S***

el resultado es el mismo: se guarda el archivo actual.

En cambio, la información de si el documento está guardado o no puede mostrarse en la **barra de estado**.

### Creación de una QAction

Podemos crear una acción desde el código y configurarla con texto, ícono, atajo de teclado, texto de estado y descripción de ayuda.

```

1 guardar = QAction(QIcon('guardar.png'), 'Guardar', self)
2 guardar.setShortcut('Ctrl+S')
3 guardar.setStatusTip('Guardar el archivo actual')
4 guardar.setToolTip('Guarda los cambios en el archivo')
5 guardar.triggered.connect(self.guardar_archivo)

```

**Nota:** `triggered` es la señal que lanza una acción cuando se ejecuta (por clic o por atajo).

Después de crearla, podemos añadirla tanto a un **menú** como a una **barra de herramientas**:

```

1 # Añadir la acción al menú
2 menu_archivo = self.menuBar().addMenu('Archivo')
3 menu_archivo.addAction(guardar)
4
5 # Añadir la acción a la barra de herramientas
6 toolbar = self.addToolBar('Archivo')
7 toolbar.addAction(guardar)

```

Y si queremos mostrar un mensaje en la **barra de estado**:

```
1 self.statusBar().showMessage("Listo")
```

### 💡 ¿Sabías que...?

Las `QAction` permiten que una misma acción esté conectada a varios elementos visuales. Si actualizas la lógica o el ícono de la acción, el cambio se refleja automáticamente en todos los lugares donde se use (menú, toolbar, etc.).

- `QMainWindow` es la base sobre la que organizamos nuestra interfaz.
- Las `QAction` son el núcleo de la interacción: permiten unificar funcionalidad.
- Los menús, barras de herramientas y barra de estado son extensiones visuales que aportan accesibilidad y contexto al usuario.

## 4.2. Barra de menú

Una de las partes más reconocibles de una aplicación de escritorio es la **barra de menú**, donde se agrupan las acciones principales organizadas en categorías (Archivo, Edición, Ver, Ayuda...).

En **Qt**, los menús se crean sobre la barra de menús de la ventana principal, accesible mediante el método `menuBar()`.

Cada menú se añade con `addMenu()`, y dentro de ellos podemos incluir **acciones** (`QAction`), **submenús** (`addMenu()`) o **separadores** (`addSeparator()`), para mantener la interfaz clara y ordenada.

### Ejemplo

Veamos cómo crear una ventana con un menú que ejecute una acción:

```
1  from PySide6.QtWidgets import QApplication, QMainWindow
2  from PySide6.QtGui import QAction, QKeySequence
3
4  # Nuestra ventana principal hereda de QMainWindow
5  class VentanaPrincipal(QMainWindow):
6      def __init__(self):
7          super().__init__()
8          self.setWindowTitle("Ventana principal con menús")
9
10     # Obtenemos la referencia a la barra de menús incluida en la clase QMainWindow
11     barra_menu = self.menuBar()
12
13     # Añadimos un menú principal
14     menu = barra_menu.addMenu("&Menú")
15
16     # Creamos una acción
17     accion = QAction("Imprimir por consola", self)
18     accion.setShortcut(QKeySequence("Ctrl+P")) # Atajo de teclado
19     accion.triggered.connect(self.imprimir_por_consola) # Vamos a definirlo más tarde
20
21     # Añadimos la acción al menú
22     menu.addAction(accion)
23
24     def imprimir_por_consola(self):
25         print("Acción lanzada desde el menú o mediante el atajo Ctrl+P")
26
27 if __name__ == "__main__":
28     app = QApplication([])
29     ventana = VentanaPrincipal()
30     ventana.show()
31     app.exec()
```

### Notas:

- **QKeySequence**: sirve para definir combinaciones de teclas (atajos) como **Ctrl+P**.
- El símbolo «**&**» delante de una letra permite **acceder al menú mediante la tecla Alt + esa letra**. Por ejemplo, si el menú se llama **&Menú**, al pulsar **Alt + M** se abrirá.

- Los **atajos** (`Ctrl + tecla`, `Alt + tecla`, etc.) se definen con `setShortcut()` y **permiten ejecutar la acción sin desplegar el menú**. En el ejemplo anterior, `Ctrl + P` ejecuta directamente la acción de imprimir por consola.

#### ► Ejercicio propuesto 10

### 4.3. Barra de herramientas

Vamos a ampliar el ejemplo anterior añadiendo la acción a una barra de herramientas. Los pasos a seguir serían los siguientes:

1. Creamos una barra de herramientas instanciando la clase **QToolBar**.
2. Añadimos la acción a la barra de herramientas con el método **addAction**.
3. Añadimos la barra de herramientas a la ventana principal con **addToolBar**.

Además del menú, podemos añadir una **barra de herramientas** con iconos para acceder rápidamente a las mismas acciones.

Estas barras pueden moverse por la ventana, acoplarse a los laterales o mostrarse/ocultarse fácilmente.

#### Ejemplo

```
1 import os
2 from PySide6.QtGui import QAction, QIcon, QKeySequence
3 from PySide6.QtWidgets import QApplication, QMainWindow, QToolBar
4
5 class VentanaPrincipal(QMainWindow):
6     def __init__(self):
7         super().__init__()
8         self.setWindowTitle("Ventana con menú y barra de herramientas")
9
10    # --- MENÚ ---
11    barra_menus = self.menuBar()
12    menu = barra_menus.addMenu("&Menú")
13
14    # Ruta del ícono
15    ruta_icone = os.path.join(os.path.dirname(__file__), "control.png")
16
17    # Acción con ícono, texto y descripción
18    accion = QAction(QIcon(ruta_icone), "Imprimir por consola")
19    accion.setWhatsThis("Imprime un texto por consola al pulsar la tecla Ctrl+P")
20    accion.setShortcut(QKeySequence("Ctrl+P"))
21    accion.triggered.connect(self.imprimir_por_consola)
22    menu.addAction(accion)
23
24    # --- BARRA DE HERRAMIENTAS ---
25    barra_herramientas = QToolBar("Barra principal")
26    barra_herramientas.addAction(accion) # Añadimos la misma acción
27    self.addToolBar(barra_herramientas)
28
29 def imprimir_por_consola(self):
30     print("Acción lanzada desde el menú, la barra de herramientas")
31
32 if __name__ == "__main__":
33     app = QApplication([])
34     ventana = VentanaPrincipal()
35     ventana.show()
36     app.exec()
```

## Comportamiento de la barra

Cuando creas una barra con `QToolBar`, Qt la trata como un **componente acoplable** dentro de la ventana principal (`QMainWindow`).

Por eso, la barra no está "fija": el usuario puede **moverla, acoplarla o ocultarla** sin que tengas que programarlo.

- **Moverla** → se puede arrastrar con el ratón por su lateral (normalmente con un puntero de "cruz") y colocarla **arriba, abajo, a la izquierda o a la derecha**.
- **Acoplarla o flotarla** → si la arrastras fuera de la ventana principal, Qt la convierte automáticamente en una **ventana flotante independiente**. Si la sueltas otra vez sobre el borde, vuelve a acoplarse.
- **Mostrar u ocultar** → si haces clic derecho sobre el área donde están las barras, Qt muestra automáticamente una lista con todas las toolbars. Puedes activar o desactivar cada una marcando su nombre.

👉 *Esto lo hace Qt por defecto, sin escribir código extra.*

### Estilo de los botones en la barra

Por defecto, los botones siguen el estilo visual del sistema operativo.

Podemos cambiarlo con el método `setToolButtonStyle()`, usando las constantes del módulo `QtCore.Qt`:

Flag	Resultado
<code>Qt.ToolButtonIconOnly</code>	Solo muestra el ícono
<code>Qt.ToolButtonTextOnly</code>	Solo muestra el texto
<code>Qt.ToolButtonTextBesideIcon</code>	Muestra el texto al lado del ícono
<code>Qt.ToolButtonTextUnderIcon</code>	Muestra el texto debajo del ícono
<code>Qt.ToolButtonFollowStyle</code>	Sigue el estilo del sistema (por defecto)

### Iconos en los menús

Por defecto, cuando una acción tiene un ícono, este **también aparece en el menú desplegable**.

A veces queda bien, pero en interfaces muy minimalistas o en macOS (donde los menús suelen ser solo texto), puede resultar visualmente recargado.

Por eso, Qt te deja **desactivar esa característica globalmente** con una línea antes de crear la ventana:

```
1 from PySide6.QtCore import Qt  
2 app = QApplication([])  
3 app.setAttribute(Qt.AA_DontShowIconsInMenus)
```

Esto mantiene los menús más limpios, aunque los iconos seguirán visibles en las barras de herramientas.

## Sincronización automática de acciones

Aquí viene una de las partes más potentes del sistema de Qt 👉

Cuando creas una `QAction` (por ejemplo, "Guardar") y la añades **a varios lugares distintos** (menú, toolbar, botón, etc.),

**Qt usa la misma instancia de acción en todos los sitios.**

Eso significa que si haces algo como esto:

```
accion_guardar.setEnabled(False)
```

👉 La acción se desactivará **en todos los lugares donde esté añadida**:

- el menú,
- la barra de herramientas,
- y cualquier otro sitio donde aparezca.

No tienes que preocuparte de sincronizar su estado a mano.

🔧 Esto también se aplica a cambios de texto, ícono, tooltip, o incluso al "checked" (si es una acción con interruptor).

## ► Ejercicio propuesto 11

## 4.4. Barra de estado

La **barra de estado** se utiliza para mostrar información útil al usuario, ya sea de forma temporal o permanente.

En Qt, accedemos a ella mediante el método `statusBar()` de `QMainWindow`.

Los métodos más usados son:

- `addWidget()` → añade un componente a la barra (por ejemplo, un `QLabel`).
- `addPermanentWidget()` → añade un componente que permanecerá visible.
- `showMessage()` → muestra un mensaje temporal.
- `clearMessage()` → borra el mensaje mostrado.

### Tipos de indicadores de estado

Cada elemento de la barra de estado puede clasificarse según su comportamiento:

#### 1. Temporal

Se muestra mientras el puntero pasa por una acción con `statusTip`, o cuando usamos `showMessage()`.

Desaparece al cumplirse el tiempo indicado o al ejecutar `clearMessage()`.

Útil para mostrar ayuda contextual (por ejemplo, "Guardar archivo actual").

#### 2. Normal

Añade componentes visibles (como `QLabel`, `QProgressBar` o `QToolButton`) a través de `addWidget()`.

Pueden ocultarse momentáneamente cuando se muestra un mensaje temporal.

Se usan para mostrar información dinámica, como "Página 3 de 10" o "Conectado".

#### 3. Permanente

Nunca se ocultan y se añaden con `addPermanentWidget()`.

Suelen emplearse para mostrar información importante, como el nombre de usuario o el estado del bloqueo de mayúsculas.

### Ejemplo

En el siguiente ejemplo, combinamos un menú, una barra de herramientas y una barra de estado con un componente permanente.

```
1 import os
2 import platform
3 from PySide6.QtGui import QAction, QIcon, QKeySequence
4 from PySide6.QtWidgets import QApplication, QMainWindow, QTool
5
6 class VentanaPrincipal(QMainWindow):
7     def __init__(self):
8         super().__init__()
9         self.setWindowTitle("Ventana principal con menú, barra
10
11         # --- MENÚ ---
12         barra_menus = self.menuBar()
13         menu = barra_menus.addMenu("&Menú")
14
15         ruta_icono = os.path.join(os.path.dirname(__file__), "imprimir.png")
16         accion = QAction(QIcon(ruta_icono), "Imprimir por consola")
17         accion.setWhatsThis("Imprime un texto por consola al principio")
18         accion.setStatusTip("Imprimir por consola")
19         accion.setShortcut(QKeySequence("Ctrl+P"))
20         accion.triggered.connect(self.imprimir_por_consola)
21         menu.addAction(accion)
22
23         # --- BARRA DE HERRAMIENTAS ---
24         barra_herramientas = QToolBar("Barra principal")
25         barra_herramientas.addAction(accion)
26         self.addToolBar(barra_herramientas)
27
28         # --- BARRA DE ESTADO ---
29         barra_estado = self.statusBar()
30
31         # Añadimos un componente permanente con el nombre del sistema
32         barra_estado.addWidget(QLabel(platform.system()))
33
34         # Mostramos un mensaje temporal (3 segundos)
35         barra_estado.showMessage("Listo. Esperando acción...", 3000)
36
37     def imprimir_por_consola(self):
38         print("Acción lanzada desde el menú, el atajo o la barra de herramientas")
39
40     if __name__ == "__main__":
41         app = QApplication([])
42         ventana = VentanaPrincipal()
43         ventana.show()
```

En este ejemplo:

- La **barra de estado** muestra un mensaje temporal al iniciar.
- El texto con `statusTip` se muestra al pasar el ratón sobre la acción.
- El **nombre del sistema operativo** aparece de forma permanente en la esquina derecha de la barra.

### 💡 ¿Sabías que...?

Si tienes varios mensajes temporales que mostrar en distintos momentos, puedes gestionarlos en cola con pequeños `QTimer`, de forma que cada mensaje se muestre durante un tiempo determinado sin superponerse.

## ► Ejercicio propuesto 12

## 4.5. Componentes flotantes (Dock Widgets)

En muchas aplicaciones de escritorio (como IDEs o editores de imágenes), hay paneles que pueden **moverse, acoplarse o flotar** libremente dentro de la ventana principal. Estos paneles se llaman **componentes flotantes** o **Dock Widgets**.

En Qt, los creamos con la clase `QDockWidget`.

Los Dock Widgets se usan para colocar **paneles auxiliares** alrededor del área principal de la aplicación.

Por ejemplo:

- Un panel lateral con herramientas.
- Una consola de mensajes.
- Un navegador de archivos.
- Una vista previa de documento.

Lo bueno es que el usuario puede:

- **Moverlos** (arrastrando su barra de título).

- **Acoplarlos** a cualquiera de los bordes de la ventana.
- **Desacoplarlos** (flotarlos como una ventana independiente).
- **Ocultarlos o mostrarlos** fácilmente con clic derecho.

### Estructura general

Un Dock Widget está formado por:

1. **Un contenedor (QDockWidget)** que puede moverse y acoplarse.
2. **Un componente interno (widget)**, que es el contenido que mostrará el dock (por ejemplo, un QTextEdit, un QLabel, una tabla, etc.).

### Ejemplo

```
1 import os
2 import platform
3 from PySide6.QtCore import Qt
4 from PySide6.QtGui import QAction, QIcon, QKeySequence
5 from PySide6.QtWidgets import (
6     QApplication, QMainWindow, QToolBar, QLabel, QDockWidget,
7 )
8
9 class VentanaPrincipal(QMainWindow):
10     def __init__(self):
11         super().__init__()
12         self.setWindowTitle("Ventana con menú, barra de herramientas y dock")
13
14         # --- MENÚ Y ACCIÓN ---
15         barra_menus = self.menuBar()
16         menu = barra_menus.addMenu("&Menú")
17
18         ruta_icone = os.path.join(os.path.dirname(__file__), "icono.png")
19         accion = QAction(QIcon(ruta_icone), "Imprimir por consola")
20         accion.setStatusTip("Imprimir por consola")
21         accion.setShortcut(QKeySequence("Ctrl+P"))
22         accion.triggered.connect(self.imprimir_por_consola)
23         menu.addAction(accion)
24
25         # --- BARRA DE HERRAMIENTAS ---
26         barra_herramientas = QToolBar("Barra principal")
27         barra_herramientas.addAction(accion)
28         self.addToolBar(barra_herramientas)
29
30         # --- BARRA DE ESTADO ---
31         barra_estado = self.statusBar()
32         barra_estado.addPermanentWidget(QLabel(platform.system()))
33         barra_estado.showMessage("Listo. Esperando acción...", 5000)
34
35         # --- COMPONENTE FLOTANTE (DOCK) ---
36         dock1 = QDockWidget("Componente base 1", self)
37         dock1.setWidget(QTextEdit(""))
38         dock1.setMinimumWidth(50)
39         self.addDockWidget(Qt.RightDockWidgetArea, dock1)
40
41         # --- COMPONENTE CENTRAL ---
42         self.setCentralWidget(QLabel("Componente principal"))
```

```

44     def imprimir_por_consola(self):
45         print("Acción lanzada desde el menú, el atajo o la bar
46
47 if __name__ == "__main__":
48     app = QApplication([])
49     ventana = VentanaPrincipal()
50     ventana.show()
51     app.exec()

```

## Qué hace este ejemplo

- La ventana principal tiene:
  - Un **menú** y una **barra de herramientas** con una acción (**Ctrl+P**).
  - Una **barra de estado** que muestra el nombre del sistema operativo.
  - Un **componente central** (una etiqueta con texto).
  - Un **dock widget lateral** con un campo de texto editable (**QTextEdit**).

## El método **addDockWidget()** y las áreas de acoplamiento

Una de las características más interesantes de los **Dock Widgets** es que podemos colocarlos en diferentes zonas de la ventana principal.

Esto se controla con el método **addDockWidget()**, que pertenece a la clase **QMainWindow**.

### Sintaxis

```
1 addDockWidget(area, dockwidget)
```

- **area** → indica en qué parte de la ventana se colocará el componente flotante.
- **dockwidget** → es el objeto **QDockWidget** que queremos añadir.

Por ejemplo:

```
1 self.addDockWidget(Qt.RightDockWidgetArea, dock1)
```

coloca el dock **dock1** en el **lado derecho** de la ventana principal.

### Zonas de acoplamiento disponibles

Qt define **cuatro áreas principales** donde se pueden colocar los Dock Widgets.

Estas áreas están representadas por constantes dentro del módulo **Qt** (en **PySide6.QtCore**).

Constante	Zona de acoplamiento	Descripción
<code>Qt.LeftDockWidgetArea</code>	Izquierda	El dock aparece en el lado izquierdo de la ventana.
<code>Qt.RightDockWidgetArea</code>	Derecha	El dock se acopla al lado derecho.
<code>Qt.TopDockWidgetArea</code>	Superior	El dock se muestra en la parte superior, sobre el contenido principal.
<code>Qt.BottomDockWidgetArea</code>	Inferior	El dock se coloca en la parte inferior de la ventana.

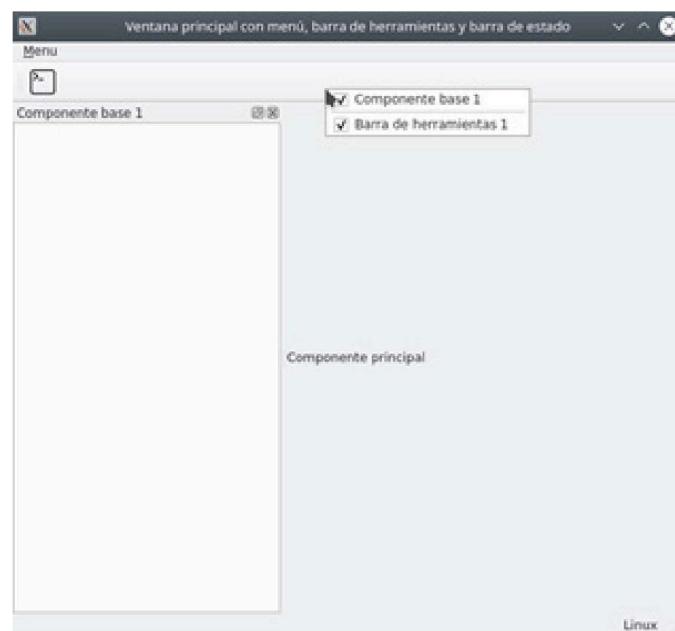
También existe una bandera combinada:

| `Qt.AllDockWidgetAreas` | Todas las zonas | Permite que el usuario pueda mover el dock a cualquier lado. |

## Comportamiento del Dock

El Dock puede:

- **Moverse** a cualquier lado (izquierda, derecha, arriba o abajo).
- **Desacoplarse** como una ventana independiente (al arrastrarlo fuera).
- **Volver a acoplarse** arrastrándolo al borde de la ventana.
- **Ocultarse o mostrarse** con clic derecho sobre el área de las barras de herramientas.



Acción del usuario	Resultado
Arrastrar por la barra de título	Mueve el panel a otro lado
Soltar fuera de la ventana principal	Se convierte en una ventana flotante
Clic derecho en la barra de herramientas	Permite ocultar o mostrar el dock
Volver a arrastrarlo dentro de la ventana	Se acopla de nuevo

Puedes añadir **varios docks** en una misma aplicación y organizarlos como quieras.

Incluso puedes **bloquearlos** para que no se muevan, con:

```
1 dock1.setFeatures(QDockWidget.NoDockWidgetFeatures)
```

O solo permitir que se **cierren o floten**:

```
1 dock1.setFeatures(QDockWidget.DockWidgetClosable | QDockWidget.D
```

Elemento	Clase Qt	Función principal
Barra de estado	<code>QStatusBar</code>	Muestra mensajes e información contextual
Widget flotante	<code>QDockWidget</code>	Paneles acoplables o flotantes para herramientas o paneles auxiliares
Mensajes temporales	<code>showMessage()</code>	Informan al usuario brevemente
Widgets permanentes	<code>addPermanentWidget()</code>	Muestran datos persistentes (por ejemplo, SO, usuario o conexión)

## ► Ejercicio propuesto 13

## 5. Diálogos y otras ventanas

Hasta ahora hemos creado aplicaciones con una única ventana. Esto es suficiente en muchos casos, pero hay situaciones en las que necesitamos mostrar otra información, pedir datos al usuario o realizar una acción extra sin recargar la ventana principal.

Para resolverlo, Qt nos permite trabajar con **diálogos** y con **otras ventanas adicionales**.

En este bloque veremos cómo utilizarlos.

### 5.1. Diálogos

Hasta este momento hemos trabajado principalmente con **QMainWindow**, que es la ventana principal de nuestras aplicaciones.

Ahora veremos cómo mostrar ventanas más pequeñas que sirven para interactuar con el usuario.

## A. QDialog

Un **QDialog** es una ventana emergente que aparece para mostrar información o pedir algún dato.

Suele abrirse cuando ocurre un evento, como pulsar un botón.

Los diálogos son **modales**, lo que significa:

- La ventana principal queda bloqueada mientras el diálogo está abierto.
- La aplicación solo continúa cuando el usuario cierra el diálogo o responde.

Como funcionan de manera independiente, necesitan su **propio bucle de eventos**.

(El uso de hilos para evitar estos bloqueos queda fuera del módulo.)

En Qt, los diálogos se crean a partir de **QDialog** o clases que hereden de ella.

Ejemplo:

```
1  from PySide6.QtWidgets import QApplication, QDialog, QMainWindow
2
3  class MainWindow(QMainWindow):
4      def __init__(self):
5          super().__init__()
6          self.setWindowTitle("Aplicación con diálogos")
7
8          boton = QPushButton("Haz clic para mostrar el diálogo")
9          boton.clicked.connect(self.mostrar_dialogo)
10         self.setCentralWidget(boton)
11
12     def mostrar_dialogo(self):
13         print("Clic recibido, se mostrará el diálogo.")
14         dialogo = QDialog(self)
15         dialogo.setWindowTitle("Ventana de diálogo")
16         dialogo.exec()
17
18     app = QApplication([])
19     ventana = MainWindow()
20     ventana.show()
21     app.exec()
```

## B. Diálogos personalizados

Hasta ahora hemos visto cómo crear un diálogo sencillo. El siguiente paso es **personalizarlo** para que muestre lo que necesitemos en cada momento de la aplicación.

Un diálogo personalizado puede incluir:

- Texto, imágenes o cualquier widget.
- Botones con distintas funciones (Aceptar, Cancelar, Guardar...).
- Un *layout* que organice los elementos según necesitemos.

Qt ofrece una colección de botones ya preparados dentro de la clase **QDialogButtonBox**.

La ventaja de usarla es que los botones se ven con el estilo propio del sistema operativo (Windows, Linux, macOS...).

Algunos de los botones disponibles son:

- **QDialogButtonBox.Ok**

- QDialogButtonBox.Cancel
- QDialogButtonBox.Save
- QDialogButtonBox.Open
- QDialogButtonBox.Close
- QDialogButtonBox.Discard
- QDialogButtonBox.Apply
- QDialogButtonBox.Reset
- QDialogButtonBox.RestoreDefaults
- QDialogButtonBox.Help
- QDialogButtonBox.SaveAll
- QDialogButtonBox.Retry
- QDialogButtonBox.Yes
- QDialogButtonBox.No
- QDialogButtonBox.YesToAll
- QDialogButtonBox.Ignore
- QDialogButtonBox.NoButton

A continuación tienes un ejemplo muy habitual: un diálogo que pregunta al usuario si quiere realizar una acción. Incluye:

- Un texto explicativo.
- Un par de botones estándar (**OK** y **Cancel**).
- Un *layout vertical* para organizar los elementos.

```
1  from PySide6.QtWidgets import (
2      QMainWindow, QApplication, QDialog, QDialogButtonBox, QVBoxLayout,
3      QLabel, QPushButton
4  )
5
6  class DialogoPersonalizado(QDialog):
7      def __init__(self, parent=None):
8          super().__init__(parent)
9          self.setWindowTitle("Diálogo personalizado")
10
11         botones = QDialogButtonBox.Ok | QDialogButtonBox.Cancel
12         caja = QDialogButtonBox(botones)
13
14         caja.accepted.connect(self.accept)
15         caja.rejected.connect(self.reject)
16
17         layout = QVBoxLayout()
18         layout.addWidget(QLabel("¿Quieres realizar esta acción?"))
19         layout.addWidget(caja)
20
21         self.setLayout(layout)
22
23 class VentanaPrincipal(QMainWindow):
24     def __init__(self):
25         super().__init__()
26         self.setWindowTitle("Aplicación con diálogo personalizado")
27
28         boton = QPushButton("Mostrar diálogo personalizado")
29         boton.clicked.connect(self.mostrar_dialogo)
30         self.setCentralWidget(boton)
31
32     def mostrar_dialogo(self):
33         print("Clic recibido, se mostrará el diálogo.")
34         dialogo = DialogoPersonalizado(self)
35         dialogo.setWindowTitle("Ventana de diálogo personalizado")
36
37         resultado = dialogo.exec()
38         if resultado:
39             print("Aceptado")
40         else:
41             print("Cancelado")
42
43 app = QApplication([])
```

```
44 | ventana = VentanaPrincipal()  
45 | ventana.show()  
46 | app.exec()
```

## Cómo recoger señales usando métodos ya predefinidos en Qt

Hasta ahora hemos creado **nuestros propios métodos** para gestionar señales.

Por ejemplo:

```
1 | boton.clicked.connect(self.mi_metodo)
```

donde `mi_metodo()` era una función escrita por nosotros.

A partir de este apartado veremos algo nuevo: **usar métodos que ya existen dentro de la clase** para responder a una señal. Esto aparece por primera vez en el ejemplo del diálogo personalizado.

### 1. ¿Qué significa usar un método predefinido?

Algunas clases de Qt incluyen métodos que ya saben cómo debe responder un diálogo ante ciertos botones.

Estos métodos se llaman normalmente `accept()` y `reject()`.

- `accept()` → Cierra el diálogo indicando que la respuesta es "Aceptar".
- `reject()` → Cierra el diálogo indicando que la respuesta es "Cancelar".

Estos métodos vienen incorporados dentro de `QDialog`, así que no tenemos que escribirlos. Podemos usarlos directamente.

### 2. ¿Por qué existen estos métodos?

Porque los diálogos son elementos muy comunes en cualquier aplicación.

Qt ya incluye:

- Comportamientos típicos.

- Señales estándar (`accepted`, `rejected`).
- Métodos que resuelven estas situaciones sin necesidad de programar más código.

### Sabías que...

`QDialog.exec()` devuelve un valor distinto según si el diálogo se cerró con `accept()` o con `reject()`.

El método `exec()` es la forma más habitual de abrir un diálogo en Qt. · A continuación...

### 3. Ventaja para el usuario

Cuando usamos un método predefinido:

- El código queda más corto y claro.
- El diálogo funciona igual en cualquier sistema operativo.
- Evitamos escribir funciones nuevas para tareas muy repetidas.

Esto no sustituye a crear métodos propios cuando la lógica lo necesita.

Solo ayuda en situaciones muy directas, como cerrar un cuadro de diálogo.

### 4. Ejemplo explicado paso a paso

```
1 caja.accepted.connect(self.accept)
2 caja.rejected.connect(self.reject)
```

- `caja.accepted` es la señal que envía el `QDialogButtonBox` cuando el usuario pulsa OK.
- `self.accept` es el método de `QDialog` que cierra el cuadro devolviendo un resultado positivo.
- `caja.rejected` se activa cuando el usuario pulsa Cancel.
- `self.reject` cierra el cuadro devolviendo un resultado negativo.

Así, el diálogo queda completamente configurado sin escribir métodos nuevos.

### 5. Para reflexionar

- ¿Podrías sustituir `self.accept` por un método propio?

- ¿En qué situaciones interesa usar un método preexistente?
- ¿Qué ocurre si añades más botones al **QDialogButtonBox**?

### Traducir los textos de los botones

Los botones estándar aparecen por defecto en inglés.

Para traducirlos podemos usar **QTranslator**:

```
1 from PySide6.QtCore import QLibraryInfo, QTranslator  
2  
3 def cargar_traductor(app):  
4     traductor = QTranslator(app)  
5     ruta = QLibraryInfo.location(QLibraryInfo.TranslationsPath)  
6     traductor.load("qt_es", ruta)  
7     app.installTranslator(traductor)  
8  
9 app = QApplication([])  
10 cargar_traductor(app)
```

### ► Ejercicio propuesto 14

## 5.2. QMessageBox

En el apartado anterior aprendimos a crear y personalizar un **QDialog**.

Qt también ofrece cuadros de diálogo ya preparados para situaciones muy comunes, como informar al usuario o pedirle una decisión rápida. Estos cuadros pertenecen a la clase **QMessageBox**.

**QMessageBox** facilita mucho el trabajo porque todo viene configurado: ícono, texto, título, botones y comportamiento básico.

Qt incluye **cuatro tipos principales**, según el tipo de mensaje:

Icono	Tipo	Cuándo usarlo
?	Question	Cuando necesitamos una respuesta del usuario.
i	Information	Para mostrar información normal.
!	Warning	Para avisos o errores no críticos.
X	Critical	Para errores graves o situaciones importantes.

Estos cuadros ya incluyen iconos y comportamiento por defecto, lo que los hace muy fáciles de usar.

### Botones disponibles

Igual que ocurre con los diálogos personalizados, `QMessageBox` también ofrece botones estándar.

Estos botones se adaptan al estilo del sistema operativo (Windows, Linux, macOS) y ya tienen un comportamiento definido.

Aquí tienes algunos de los más usados:

- `QMessageBox.Ok`
- `QMessageBox.Cancel`
- `QMessageBox.Yes`
- `QMessageBox.No`
- `QMessageBox.Retry`
- `QMessageBox.Close`
- `QMessageBox.Ignore`
- `QMessageBox.Discard`
- `QMessageBox.Save`
- `QMessageBox.Open`

En la práctica, basta con saber usar los básicos (Ok, Cancel, Yes, No), aunque es bueno conocer el resto para casos más específicos.

### Ejemplo: mensaje crítico

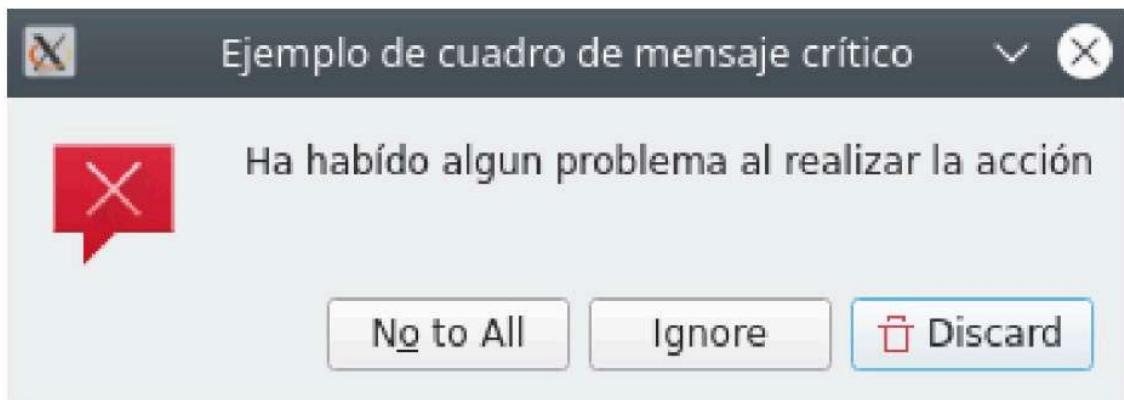
Este es un ejemplo típico en muchas aplicaciones: mostrar un mensaje de error importante y pedir al usuario que elija cómo continuar.

```
1  from PySide6.QtWidgets import (
2      QApplication, QMessageBox, QMainWindow, QPushButton
3  )
4
5  class VentanaPrincipal(QMainWindow):
6      def __init__(self):
7          super().__init__()
8          self.setWindowTitle("Aplicación con mensaje crítico")
9          boton = QPushButton("Haz clic para ver el mensaje crítico")
10         boton.clicked.connect(self.mostrar_dialogo)
11         self.setCentralWidget(boton)
12
13     def mostrar_dialogo(self):
14         # Cuadro de mensaje crítico
15         boton_pulsado = QMessageBox.critical(
16             self,
17             "Ejemplo de cuadro de mensaje crítico",
18             "Ha ocurrido un problema al realizar la acción",
19             buttons=QMessageBox.Discard | QMessageBox.NoToAll |
20             defaultButton=QMessageBox.Discard
21         )
22
23         # Comprobamos qué botón ha seleccionado el usuario
24         if boton_pulsado == QMessageBox.Discard:
25             print("Descartado")
26         elif boton_pulsado == QMessageBox.NoToAll:
27             print("No a todo")
28         else:
29             print("Ignorado")
30
31     app = QApplication([])
32     ventana = VentanaPrincipal()
33     ventana.show()
34     app.exec()
```



### Explicación de boton\_pulsado

Este fragmento crea y muestra un cuadro de mensaje crítico: · Snippet · Vamos lí...



Este tipo de cuadro aparece en muchas aplicaciones cuando algo no funciona como se esperaba. Lo habitual es ofrecer varias opciones para que el usuario decida:

- **Descartar la operación.**
- **Aplicar “No a todo”** si se repite la misma acción en varios elementos.
- **Ignorar temporalmente el error.**

Qt gestiona la creación del cuadro, los iconos, la posición de los botones y el estilo general. Solo necesitamos indicar qué opciones queremos mostrar y comprobar qué botón ha elegido el usuario.

### Traducir los textos al español

Los botones de `QMessageBox` suelen aparecer en inglés.

Para traducirlos podemos utilizar `QTranslator`, igual que en el apartado anterior.

```
1 def cargar_traductor(app):
2     traductor = QTranslator(app)
3     ruta = QLibraryInfo.location(QLibraryInfo.TranslationsPath)
4     traductor.load("qt_es", ruta)
5     app.installTranslator(traductor)
6
7 app = QApplication([])
8 cargar_traductor(app)
```

Esta función debe ejecutarse antes de crear la ventana principal para que todos los mensajes aparezcan ya traducidos.

### ► Ejercicio propuesto 15

## 5.3. Otros diálogos

Además de `QDialog` y `QMessageBox`, Qt ofrece varios diálogos ya preparados para tareas muy comunes en cualquier aplicación.

La ventaja es que **no hay que construirlos desde cero**: basta con llamarlos y recoger el valor que indique el usuario.

Los diálogos más comunes que encontrarás en **QtWidgets** son:

- **QColorDialog** → seleccionar un color
- **QFileDialog** → elegir archivos o carpetas
- **QFontDialog** → seleccionar una fuente
- **QInputDialog** → pedir un dato simple

A continuación tienes ejemplos sencillos de cada uno.

### Ejemplo 1. Seleccionar un archivo (abrir)

Este diálogo permite elegir un archivo del sistema.

Qt devuelve una tupla: el primer valor es la ruta del archivo seleccionado y el segundo el filtro aplicado. Por ejemplo:

```
( '/home/usuario/documento.txt' , ' Documentos de texto (*.txt)' )
```

```
1 def mostrar_dialogo(self):  
2     ventana_dialogo = QFileDialog.getOpenFileName(  
3         self,  
4         caption="Abrir archivo ...",  
5         dir=". /",  
6         filter="Documentos de texto (*.txt);;Documentos PDF (*.pdf)",  
7         selectedFilter="Documentos de texto (*.txt)"  
8     )  
9  
10    archivo = ventana_dialogo[0] # Ruta del archivo seleccionado
```

### ¿Cuándo usarlo?

Para cargar documentos, imágenes, bases de datos, etc.

### Nota sobre los filtros

Cuando usamos QFileDialog para abrir o guardar archivos, podemos definir filtr...

## Ejemplo 2. Seleccionar un archivo para guardar

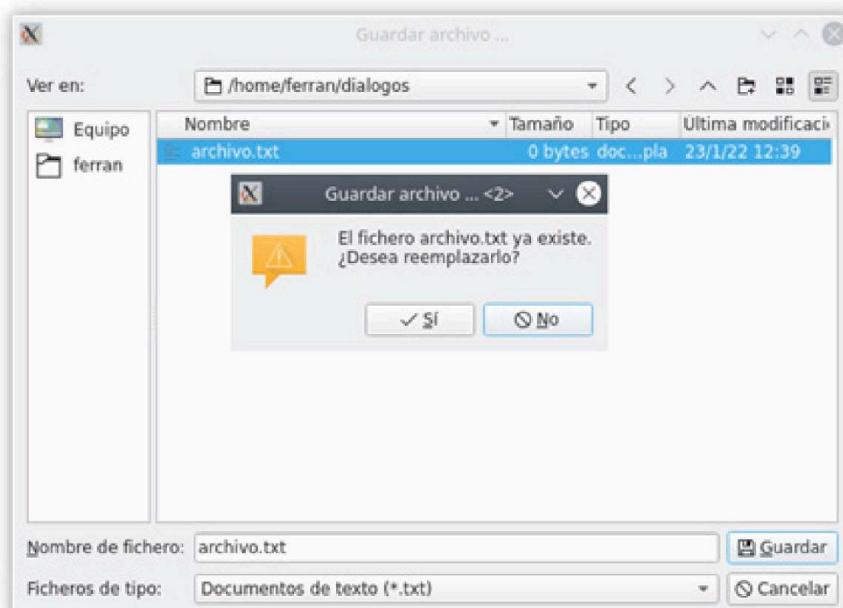
Funciona igual que el anterior, pero permite elegir el nombre de un archivo nuevo.

Si ya existe, Qt pregunta si queremos sobrescribirlo.

```

1 def mostrar_dialogo(self):
2     ventana_dialogo = QFileDialog.getSaveFileName(
3         self,
4         caption="Guardar archivo ...",
5         dir="./",
6         filter="Documentos de texto (*.txt);;Documentos PDF (*.pdf)",
7         selectedFilter="Documentos de texto (*.txt)"
8     )
9
10    archivo = ventana_dialogo[0]
11    print(archivo)

```



## Ejemplo 3. Seleccionar un color

**QColorDialog** muestra una ventana para seleccionar un color.

Devuelve un objeto `QColor`.

```

1 def mostrar_dialogo(self):
2     color = QColorDialog.getColor()
3     if color.isValid():
4         self.boton.setStyleSheet(f"background-color: {color.name()}")

```



### Detalles importantes:

- Si el usuario pulsa "Aceptar", `isValid()` será `True`.
- `color.name()` devuelve el color en formato hexadecimal (por ejemplo, `#ff0000`).

### Uso típico:

Opciones de personalización, cambios de tema, ajustes visuales.

### Nota:

En PySide6, `QColorDialog` se importa desde `QtWidgets` y se puede usar sin necesidad de importar `QColor`, porque el diálogo ya devuelve un objeto `QColor`.

Solo hay que importar `QColor` cuando el programa crea colores manualmente o realiza operaciones avanzadas con ellos.

## Ejemplo 4. Seleccionar una fuente

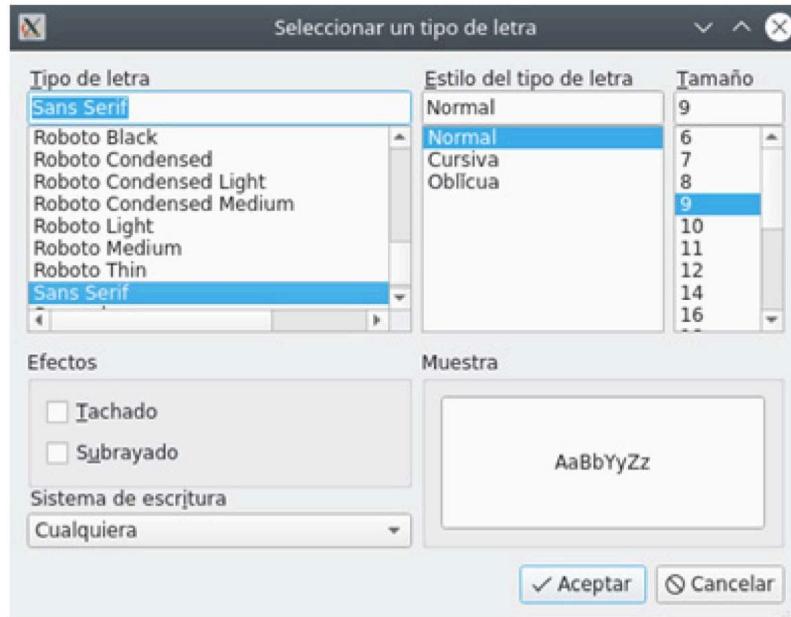
El método `QFontDialog.getFont()` devuelve dos valores:

- `seleccionada`: si el usuario ha pulsado OK
- `fuente`: la fuente elegida

```

1 def mostrar_dialogo(self):
2     seleccionada, fuente = QFontDialog.getFont(self)
3     if seleccionada:
4         self.boton.setFont(fuente)

```



### Muy útil cuando:

El programa permite personalizar texto, editor de notas, aplicación de escritura o chat.

### Nota:

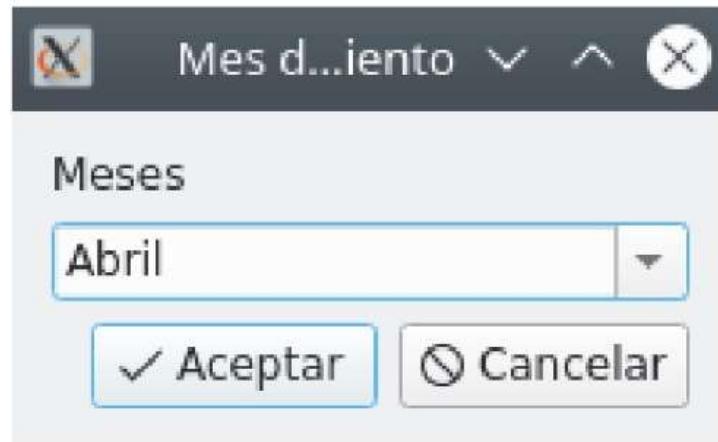
`QFont` se importa desde `PySide6.QtGui` porque en PySide6 las clases relacionadas con gráficos, colores y tipografías no están dentro de `QtWidgets`. Si el programa usa objetos de tipo `QFont` (por ejemplo, al aplicar una fuente elegida en un diálogo), es necesario importar esta clase.

### Ejemplo 5. Obtener un dato rápido (`QInputDialog`)

Este diálogo es perfecto para pedir datos sencillos sin crear un formulario completo.

Ejemplo: elegir un mes de nacimiento y mostrarlo por consola.

```
1 def mostrar_dialogo(self):  
2  
3     mes, seleccionado = QInputDialog.getItem(  
4         self,  
5         "Mes de nacimiento",  
6         "Meses",  
7         ["Enero", "Febrero", "Marzo", "Abril", "Mayo",  
8          "Junio", "Julio", "Agosto", "Septiembre",  
9          "Octubre", "Noviembre", "Diciembre"]  
10    )  
11  
12    if seleccionado:  
13        print(mes)
```



### También permite pedir:

- Un texto → `getText()`
- Un número entero → `getInt()`
- Un número decimal → `getDouble()`

Es muy común en configuraciones rápidas, ajustes o formularios simples.

### ► Ejercicio propuesto 16

## 5.4. Otras ventanas

Hasta ahora hemos trabajado con **diálogos modales**, que bloquean la aplicación mientras están abiertos.

Sin embargo, hay situaciones en las que queremos abrir otra ventana **sin bloquear la principal**. Por ejemplo:

- Mostrar información adicional
- Abrir un panel de herramientas
- Visualizar datos en paralelo
- Crear dos interfaces independientes

En Qt esto es muy sencillo: **cualquier widget que no tenga un parent se convierte automáticamente en una ventana**.

Esto significa que, para crear una nueva ventana, basta con:

1. Crear el widget (por ejemplo, un QLabel, un QWidget, etc.)
2. No darle un parent
3. Llamar a .show()

Incluso podríamos tener varias ventanas principales (**QMainWindow**) abiertas a la vez.

### Ejemplo: aplicación con dos ventanas

En este ejemplo tenemos:

- Una **ventana principal** con un botón.
- Una **ventana secundaria** que aparece u oculta según lo que pulse el usuario, es decir, la ventana secundaria **no se crea al inicio**, solo cuando se pulsa el botón.
  - Si la ventana secundaria está abierta, el mismo botón permite **ocultarla**.
  - Si está oculta, el botón vuelve a **mostrarla**.

```
1  from PySide6.QtWidgets import (
2      QApplication, QMainWindow, QPushButton, QLabel
3  )
4
5  # Ventana secundaria que hereda de QLabel.
6  # Al no tener parent, aparece como una ventana independiente.
7  class OtraVentana(QLabel):
8      def __init__(self):
9          super().__init__()
10         self.setText("La otra ventana")
11
12 class VentanaPrincipal(QMainWindow):
13     def __init__(self):
14         super().__init__()
15         self.setWindowTitle("Aplicación con dos ventanas")
16
17         # Guardamos la referencia de la ventana secundaria
18         self.otra_ventana = None
19
20         self.boton = QPushButton("Mostrar/ocultar otra ventana")
21         self.boton.clicked.connect(self.mostrar_otra_ventana)
22         self.setCentralWidget(self.boton)
23
24     def mostrar_otra_ventana(self):
25         # Si todavía no existe, la creamos
26         if self.otra_ventana is None:
27             self.otra_ventana = OtraVentana()
28             self.otra_ventana.show()
29         else:
30             # Si existe pero está oculta, la mostramos
31             if self.otra_ventana.isHidden():
32                 self.otra_ventana.move(self.pos())
33                 self.otra_ventana.show()
34             # Si está visible, la ocultamos
35             else:
36                 self.otra_ventana.hide()
37
38 app = QApplication([])
39 ventana_principal = VentanaPrincipal()
40 ventana_principal.show()
41 app.exec()
```

## Notas

- Si la ventana se guarda en una **variable local**, Python la destruye al salir del método. Por eso se guarda como **atributo de la clase** (`self.otra_ventana`).
- La ventana secundaria **no bloquea** la principal: ambas pueden usarse a la vez.
- Puedes crear tantas ventanas independientes como necesites. Por ejemplo: múltiples editores, paneles, herramientas flotantes, etc.

### ► Ejercicio propuesto 17