IIC2233 • PROGRAMACIÓN AVANZADA

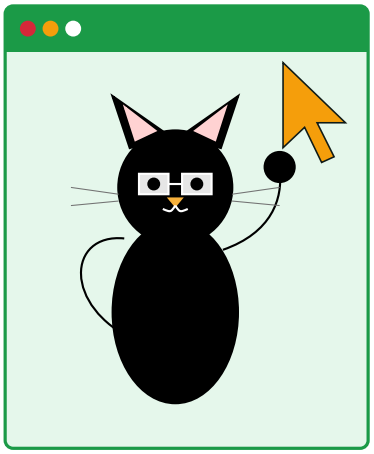
INTERFACES GRÁFICAS PyQt6

`QWidget` • layouts • señales y *slots*

front-end y *back-end* • varias ventanas

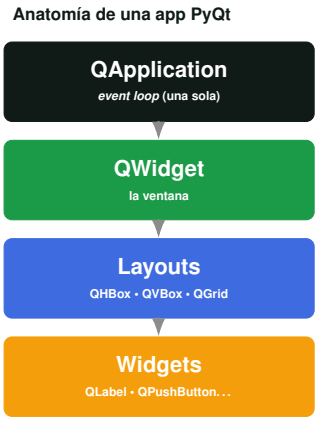
de la consola a una aplicación que reacciona a lo que haces



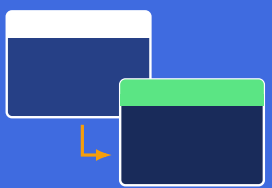


Felipín quiso hacer una app con ventanas. Leyó todo con `input()` en la consola, abrió una segunda ventana guardándola en una variable local (¡Python la borró apenas terminó el método!) y metió la lógica de ordenar dentro de la clase de la ventana. Ahora cada cambio rompe tres cosas… y el programa "se cae" sin mostrar ningún error.

No seas como Felipín. En este resumen aprenderás a crear **ventanas**, ubicar **widgets** con **layouts**, reaccionar a **eventos** con **señales y *slots***, y separar el ***front-end*** del ***back-end***.







Resumen de Contenidos

IIC2233 — Programación Avanzada • Edición Ventana abierta

Resumen elaborado por el equipo de ayudantía, en base al material oficial del curso

Tabla de Contenidos

1 Introducción 2

2 Interfaces gráficas y PyQt 3

3 Tu primera ventana 5

4 Elementos gráficos básicos 8

5 Eventos y señales 15

6 Diseño de software: *front-end* y *back-end* 22

7 Conexión entre múltiples ventanas 28

8 Hoja de referencia 31



1 Introducción

Hasta ahora nuestros programas conversaban con el usuario por la **consola**: pedían un dato con `input()`, hacían algo y mostraban un resultado con `print()`. Este resumen reúne el contenido de la semana de **Interfaces Gráficas** de IIC2233: cómo construir **ventanas**, botones y cuadros de texto con **PyQt6**, cómo hacer que el programa **reaccione** a lo que hace el usuario (clics, teclas, movimiento del *mouse*), cómo **organizar** el código para que no se vuelva un enredo (*front-end* y *back-end*) y cómo comunicar **varias ventanas** entre sí.

■ Los cinco bloques de este resumen

- **Fundamentos de GUI y PyQt** (Secciones 2 y 3): qué es una interfaz gráfica, *polling* vs. eventos, módulos de PyQt, `QApplication`, `QWidget` y el *main loop*.
- **Widgets y *layouts*** (Sección 4): sistema de coordenadas, `QLabel`, `QLineEdit`, `QPixmap`, `QPushButton`, `QHBoxLayout`, `QVBoxLayout`, `QGridLayout` y clases propias con POO.
- **Eventos y señales** (Sección 5): *signals* y *slots*, `sender()`, eventos de *mouse* y teclado, y señales personalizadas con `pyqtSignal`.
- **Diseño *front-end*/*back-end*** (Sección 6): cohesión, acoplamiento y cómo las señales los separan.
- **Varias ventanas** (Sección 7): por qué una ventana "desaparece" y cómo conectarlas con señales.

★ Cómo leer este resumen

Los recuadros **rojos** marcan errores típicos y trampas; los **ámbar** son definiciones importantes; los **azules** son analogías para fijar ideas; y los recuadros **"Qué verás"** describen lo que aparece en pantalla al ejecutar el ejemplo. Una idea central se repite en todo el material: *una interfaz gráfica no sigue una línea recta, se queda esperando eventos y reacciona a ellos*.

! Ejecuta los ejemplos en tu computador, no en Jupyter

Jupyter no fue creado para ejecutar interfaces de escritorio, así que los ejemplos de este resumen se prueban como **scripts** `.py` desde tu propia terminal. Asegúrate de que la terminal esté posicionada **en el mismo directorio** donde está el archivo que vas a ejecutar (por ejemplo, para cargar la carpeta `img/`).



el **ciclo completo** de una interfaz gráfica: el usuario actúa, el programa reacciona

2 Interfaces gráficas y PyQt

2.1 ¿Qué es una GUI?

Una **GUI** (*Graphical User Interface*, o **interfaz gráfica de usuario**) es una forma de interactuar con el computador mediante abstracciones gráficas: ventanas, íconos, menús, hipertexto y un dispositivo que mueve un puntero en pantalla, el *mouse*. Las GUI nacieron en los años 1960 en el *Stanford Research Institute* (SRI), con un grupo liderado por **Doug Engelbart**, y se presentaron en una demostración histórica llamada *The Mother of All Demos*. Su influencia llegó a Xerox PARC (con el computador Xerox Alto) y, de ahí, a los sistemas operativos de los computadores personales. Engelbart recibió el *Turing Award* en 1997 por su visión de la computación interactiva.



Hoy, para la mayoría de los usuarios una GUI es la forma más natural de usar un programa: formularios web, escritorios de los sistemas operativos, menús desplegables en apps de escritorio y de celular. Con ellas el programa puede **pedir parámetros**, **mostrar resultados** y **ofrecer controles** de manera mucho más amigable que una consola.

2.2 Polling vs. arquitectura basada en eventos

Un programa de consola sigue **un único flujo**. Si quisiéramos hacer una interfaz con ese estilo, tendríamos que ejecutar para siempre un ciclo que le pregunte a cada elemento si debe hacer algo:

```
while True:
    # Revisar si hay entrada desde el teclado
    # Revisar si hay entrada desde el mouse
    # Revisar si hay acción sobre la ventana 1 y procesarla
    # Revisar si hay acción sobre la ventana 2 y procesarla
    ...
```

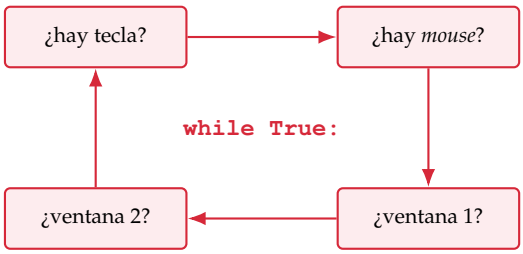
Este modo de operación se llama ***polling***: revisar activamente cada elemento de la interfaz para saber si hay algo que hacer. Es **costoso**, porque la mayoría de las veces la respuesta es "nada". Las aplicaciones modernas usan, en cambio, una **arquitectura basada en manejo de eventos**.

Evento y *handler*

Un **evento** es una acción que ocurre en el programa, gatillada por el usuario o por otra parte del programa, y a la cual se le puede definir un comportamiento. Para cada evento `e` definimos una función `e_handler` (**manejador**) que se ejecuta cada vez que `e` ocurre. El programa detecta el evento y ejecuta el *handler* **inmediatamente**, sin esperar a que otras cosas terminen: se dice que los eventos se manejan de forma **asíncrona**, es decir, no necesariamente por el flujo principal del programa.

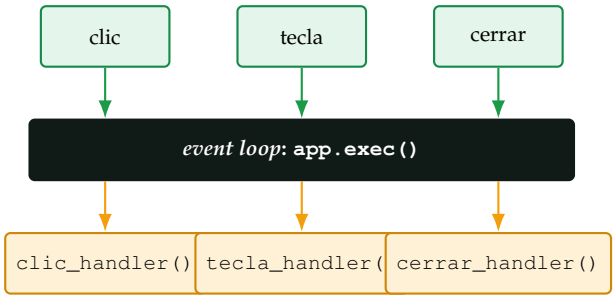
Ejemplos de eventos: el usuario hizo clic en el botón 1; abrió o cerró una ventana; presionó una tecla; el *mouse* está sobre una imagen.

Polling: preguntar a todos, todo el tiempo



casi siempre la respuesta es "nada", pero
la CPU trabaja de todos modos

Eventos: esperar y reaccionar



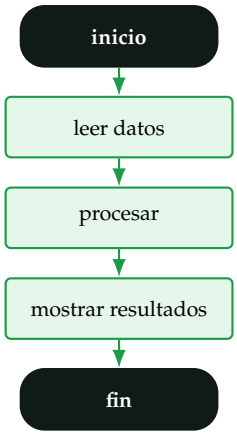
el programa espera; cuando ocurre algo,
el *loop* ejecuta el *handler* que corresponde

★ Analogía de restaurante

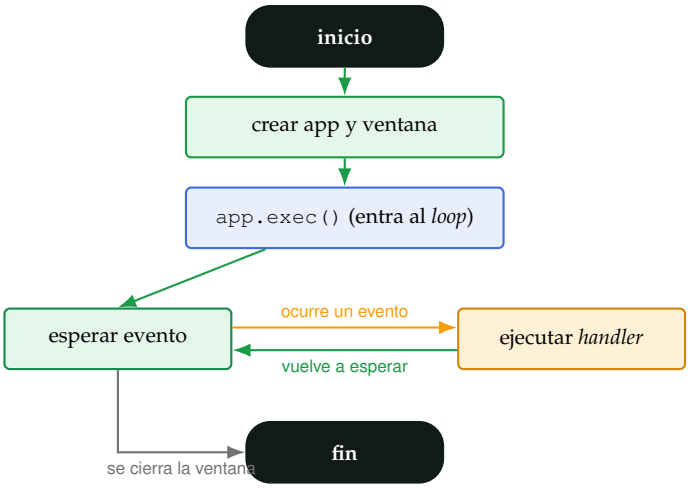
Polling es un garzón que recorre **todas las mesas una y otra vez** preguntando "¿desea algo?". La arquitectura de eventos es un restaurante donde cada mesa tiene un **timbre**: el garzón espera tranquilo y solo actúa cuando alguien lo toca. El timbre es el *evento*; atender esa mesa es el *handler*.

2.2.1 Flujo lineal vs. flujo basado en eventos

Programa lineal (consola)



GUI basada en eventos

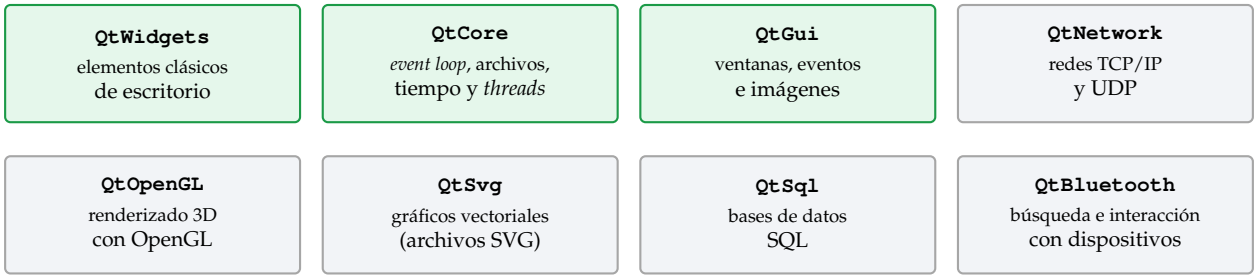


¿Cómo leerlo? En el programa lineal, **tú** decides el orden de los pasos. En una GUI, el orden lo decide **el usuario**: tu código define *qué hacer* ante cada evento, y el *loop* se queda esperando hasta que alguno ocurra. El *loop* empieza recién cuando se llama a `exec()` de `QApplication`.

2.3 Frameworks y PyQt

Existen muchos *frameworks* para crear GUI de escritorio (cada uno se adapta a distintos sistemas operativos: Ubuntu, macOS, Windows, Android…). Un ***framework*** es un entorno de desarrollo, o conjunto de reglas estandarizadas, para crear algo; en este caso, interfaces. También hay un campo completo de interfaces para la web, con sus propios patrones, pero en su mayoría siguen la misma arquitectura de eventos.

En el curso usamos **PyQt**, y específicamente **PyQt6**: un *framework* **multiplataforma** (funciona en varios sistemas operativos) basado en la biblioteca de C++ **Qt**. Se organiza en **módulos**, cada uno con una funcionalidad distinta:



verde: módulos que usamos en el curso **gris**: existen, pero no los veremos

Módulo	Clases que aparecen en este resumen
`QtWidgets`	`QApplication`, `QWidget`, `QLabel`, `QLineEdit`, `QPushButton`, `QHBoxLayout`, `QVBoxLayout`, `QGridLayout`
`QtGui`	`QPixmap` (imágenes), `QMouseEvent`, `QKeyEvent` (información de los eventos)
`QtCore`	`pyqtSignal`, `QObject`, `QCoreApplication`

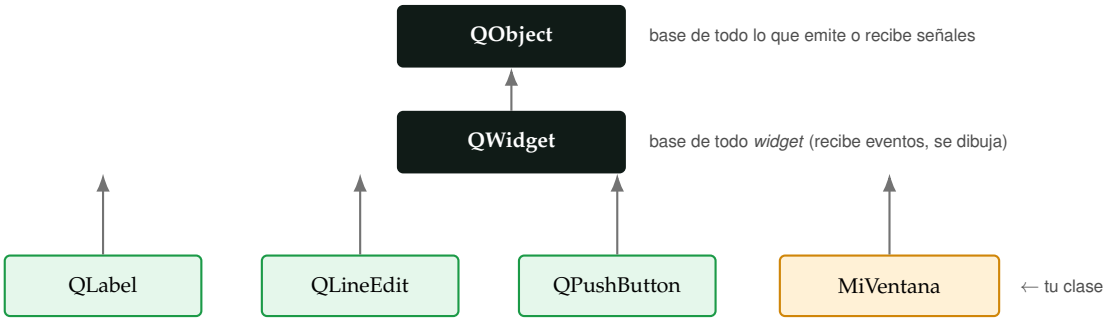
★ Instalación

PyQt6 se instala como cualquier librería externa de Python (por ejemplo, con `pip install PyQt6`). Las instrucciones oficiales del curso están en la **wiki del Syllabus**, sección *Software Obligatorio*; si tienes problemas, escribe en las *discussions* del curso. La documentación completa de PyQt6 está en el sitio de Riverbank Computing.

3 Tu primera ventana

3.1 Widgets: la pieza básica

En PyQt, los elementos que **reciben eventos** y que se **dibujan en pantalla** se llaman *widgets*. Una ventana, un botón, una etiqueta y un cuadro de texto son todos *widgets*. La clase base de todos ellos es `QWidget` (del módulo `QtWidgets`): puede recibir los eventos del sistema (*mouse*, teclado...) y mostrar una representación de sí misma. Todos los *widgets* son **rectangulares**.



las flechas significan "hereda de". Jerarquía simplificada: PyQt tiene capas intermedias que aquí no necesitamos

Como `QWidget` es a su vez un `QObject`, todo *widget* **puede definir señales** (lo veremos en la Sección 5). Y como son clases, podemos **heredar** de ellas y aplicar toda la POO que conocemos.

3.2 La aplicación: `QApplication`

Antes de crear cualquier ventana hay que crear la **aplicación** que las contendrá, con la clase `QApplication`. Esta clase contiene el ***loop* de eventos** (*main loop*) y se encarga, entre otras cosas, de inicializar y cerrar los *widgets*.

■ Reglas de `QApplication`

- Se instancia **antes** que cualquier otro *widget*.
- Existe **una sola instancia** por aplicación, sin importar cuántas ventanas tenga.

- Recibe como argumento los parámetros de la línea de comandos. Como casi nunca los usaremos, se instancia con una **lista vacía**: `QApplication([])`.

3.3 Una ventana vacía

Este es el programa mínimo que muestra una ventana (archivo `1-interfaces-graficas/ventana-simple.py`):

```python
import sys
from PyQt6.QtWidgets import QWidget, QApplication


class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()

        # Definimos la geometría de la ventana.
        # Parámetros: (x_superior_izq, y_superior_izq, ancho, alto)
        self.setGeometry(200, 100, 300, 300)

        # Podemos dar nombre a la ventana (opcional).
        self.setWindowTitle('Mi Primera Ventana')


if __name__ == '__main__':
    def hook(type_, value, traceback) -> None:
        # Imprimimos el error con el hook original de Python. Al instalar
        # nuestro propio sys.excepthook, PyQt no cierra el programa
        sys.__excepthook__(type_, value, traceback)

    sys.excepthook = hook

    app = QApplication([])  # Creamos la base de la app: QApplication.
    ventana = MiVentana()   # Construimos un QWidget que será nuestra ventana.
    ventana.show()          # Mostramos la ventana.
    codigo = app.exec()     # Inicia el loop de eventos y retorna un código de salida.
    print(f"Exit code: {codigo}")
    sys.exit(codigo)        # Termina el programa con el código de salida de la app
                            # (0 significa que terminó sin errores).
```

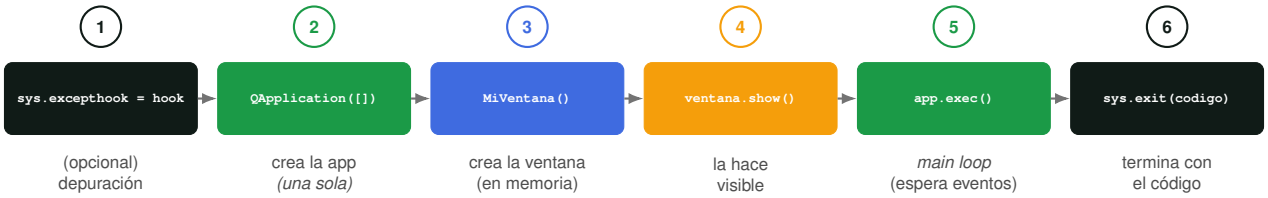
Qué verás



Una ventana vacía con el título “Mi Primera Ventana”. Su aspecto exacto (bordes, botones de cerrar) depende de tu sistema operativo: Windows, macOS y Linux dibujan la decoración de la ventana a su manera, pero el contenido es el mismo.

3.3.1 Línea por línea

Código	Qué hace
`class MiVentana(QWidget)`	Nuestra ventana **es un** `QWidget`: hereda todo su comportamiento.
`super().__init__()`	Inicializa la parte de `QWidget`. Sin esto la ventana no funciona.
`setGeometry(200, 100, 300, 300)`	Posición **y** tamaño en un solo paso: $(x, y,$ ancho, alto$)$.
`setWindowTitle('...')`	Pone el título de la ventana (opcional).
`app = QApplication([])`	Crea la aplicación y su *main loop* (todavía no se ejecuta).
`ventana = MiVentana()`	Crea el objeto. Hasta aquí la ventana **solo existe en memoria**.
`ventana.show()`	La muestra en pantalla.
`app.exec()`	**Arranca el *main loop***: desde aquí se detectan los eventos del sistema. Retorna un código de salida cuando se cierra la app.
`sys.exit(codigo)`	Termina el programa con ese código (`0` = sin errores).



3.4 Qué pasa si...

Si...	Entonces...
comentas `app = QApplication([])`	El programa **falla**: no hay una aplicación instanciada antes de crear un *widget*.
duplicas `QApplication([])`	También falla: habría **dos** instancias, y solo puede haber una.
olvidas `ventana.show()`	La ventana existe en memoria, pero **nunca se ve**.
olvidas `app.exec()`	No hay *main loop*: el programa **termina de inmediato**.

3.5 *Debuggeo* en PyQt

Con PyQt es común que el programa **se caiga sin mostrar ningún error** en pantalla, y entonces no tenemos pistas de qué ocurrió. Para no llenar el código de `print`, el material propone instalar un ***hook*** de excepciones: reemplazamos `sys.excepthook` por una función que imprime el error de origen usando el *hook* original de Python (`sys.__excepthook__`). Al hacer esto, **PyQt no cierra el programa** y tú ves el *traceback* en la terminal.

```
if __name__ == '__main__':
    def hook(type_, value, traceback) -> None:
        sys.__excepthook__(type_, value, traceback)

    sys.excepthook = hook

    app = QApplication([])
    ventana = MiVentana()
    ventana.show()
    sys.exit(app.exec())
```

★ Plantilla base

Este bloque `if __name__ == '__main__'` es la **plantilla de arranque** de casi todos los ejemplos del resumen. Por brevedad, de aquí en adelante a veces mostraremos solo las clases o una versión corta del `main`; en tus programas conviene mantener siempre el *hook*.

✓ Resumen express: fundamentos

- Una GUI funciona con **eventos**: el programa espera en un *loop* y ejecuta *handlers*. El *polling* (preguntar siempre a todos) es costoso.
- PyQt6 es un *framework* multiplataforma basado en Qt; sus módulos principales aquí son `QtWidgets`, `QtGui` y `QtCore`.
- Todo *widget* hereda de `QWidget`; una ventana es un `QWidget` que tú extiendes.
- Receta mínima: `QApplication([])` (una sola, antes de todo) → crear la ventana → `show()` → `app.exec()` → `sys.exit(...)`.
- Instala un `sys.excepthook` para ver los errores que PyQt esconde.

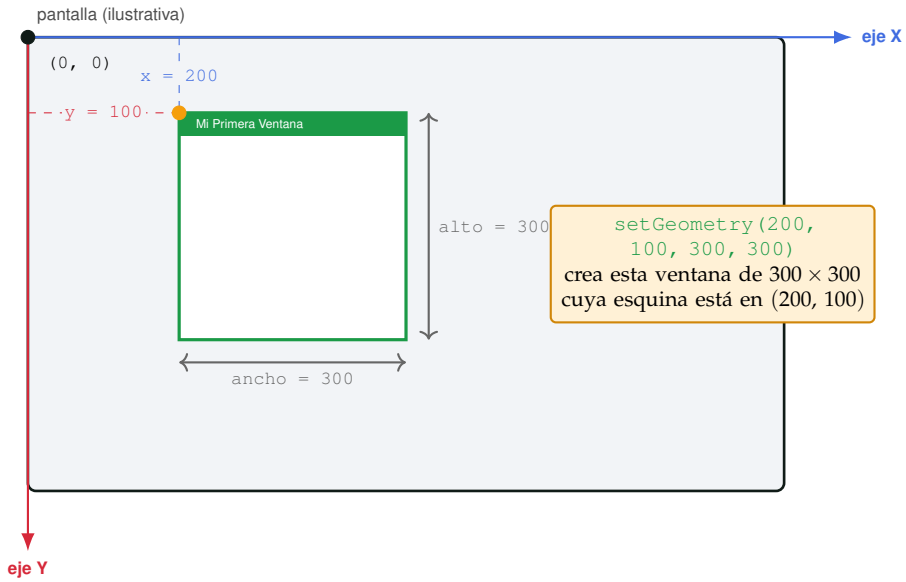
4 Elementos gráficos básicos

4.1 El sistema de coordenadas

Antes de agregar elementos a una ventana hay que entender cómo PyQt **posiciona** las cosas. En el ejemplo anterior usamos `self.setGeometry(200, 100, 300, 300)`. Sus cuatro parámetros son, en orden:

Parámetro	Significado
`x`	posición horizontal de la esquina superior izquierda
`y`	posición vertical de la esquina superior izquierda
`ancho`	ancho del rectángulo
`alto`	alto del rectángulo

Con esos cuatro valores queda definida por completo una ventana rectangular. El detalle importante es el **origen**: el punto $(0, 0)$ está en la **esquina superior izquierda de la pantalla**, el eje **X crece hacia la derecha** y el eje **Y crece hacia abajo** (al revés de lo que se dibuja en matemáticas).

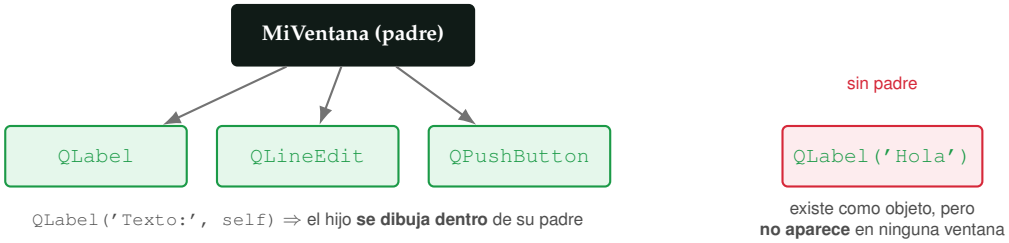


★ Pruébalo

Cambia la línea a `self.setGeometry(300, 200, 300, 300)` y verás la ventana aparecer más lejos de la esquina superior izquierda de tu pantalla ($(300, 200)$ en vez de $(200, 100)$). Juega con los cuatro valores para ver cómo cambian posición y tamaño.

4.2 *Widgets* hijos y *parent*

Los *widgets* que agregas dentro de una ventana (etiquetas, botones, cuadros de texto) **deben pertenecer a otro *widget***, su ***parent*** o *widget* padre. Por eso, al crearlos se les entrega `self`: la ventana que los contiene.



Además, la posición de un hijo se mide con **el mismo sistema de coordenadas, pero con origen en la esquina superior izquierda de su ventana padre**, no de la pantalla.

4.3 Etiquetas y cuadros de texto

PyQt tiene *widgets* para la entrada y salida de información. Las **etiquetas** (`QLabel`) muestran texto estático o dinámico; los **cuadros de texto** (`QLineEdit`) reciben texto del usuario. Ambos se crean con `Clase(texto_inicial, padre)` (archivo `2-elementos-graficos-basicos/1_label-y-lineedit.py`):

```python
import sys
from PyQt6.QtWidgets import QApplication, QWidget, QLabel, QLineEdit


class MiVentana(QWidget):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        # Llamamos a un método propio que inicializa los elementos.
        self.init_gui()

    def init_gui(self) -> None:
        # Ajustamos la geometría de la ventana y su título.
        self.setGeometry(200, 100, 200, 300)
        self.setWindowTitle('Ventana con label y cuadro de texto')

        # QLabel(texto_inicial, padre). Luego las ubicamos en la ventana.
        self.label1 = QLabel('Texto:', self)
        self.label1.move(10, 15)

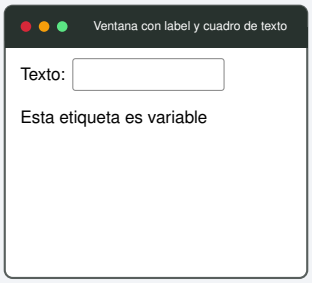
        self.label2 = QLabel('Esta etiqueta es variable', self)
        self.label2.move(10, 50)

        # QLineEdit(texto_inicial, padre). Definimos posición y porte.
        self.edit = QLineEdit('', self)
        self.edit.setGeometry(45, 15, 100, 20)

        # Con todos los elementos agregados, mostramos la ventana.
        self.show()


if __name__ == '__main__':
    app = QApplication([])
    ventana = MiVentana()
    sys.exit(app.exec())
```

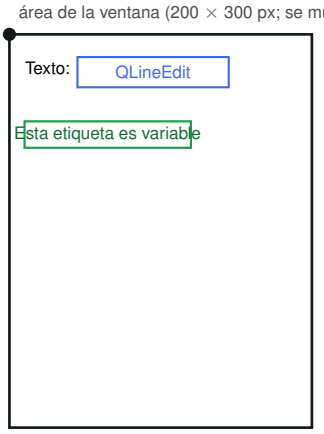
Qué verás



Dos etiquetas y un cuadro de texto. La segunda etiqueta es "variable" porque más adelante podremos cambiar su texto desde el código.

¿Qué hace este código? El método `init_gui` (llamado desde `__init__`) crea y posiciona los *widgets*. `move(x,`

`y)` cambia solo la posición; `setGeometry(x, y, ancho, alto)` cambia posición y tamaño. Separar el armado de la interfaz en un método `init_gui` mantiene `__init__` ordenado y es el patrón que seguiremos.




```
label1.move(10, 15)
edit.setGeometry(45, 15, 100, 20)

label2.move(10, 50)
```

Dentro de la ventana el origen $(0,0)$ está en **su** esquina superior izquierda. Escala del dibujo: 1 cm = 50 px.

★ Qué pasa si no entregas `self`

Prueba crear los `QLabel` y `QLineEdit` **sin** pasarles `self`: no aparecerán. Se crean los objetos, pero sin un padre al que pertenezcan no se visualizan.

4.4 Imágenes

Para mostrar una imagen se usa `QPixmap` (del módulo `QtGui`), que carga un conjunto de píxeles desde un archivo. Para dibujarlo en la ventana, esos píxeles se cargan dentro de un `QLabel` con `setPixmap` (archivo `2_pixmap.py`):

```python
import os
from PyQt6.QtWidgets import QApplication, QWidget, QLabel
from PyQt6.QtGui import QPixmap


class MiVentana(QWidget):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.init_gui()

    def init_gui(self) -> None:
        self.setGeometry(200, 100, 200, 200)
        self.setWindowTitle('Ventana con imagen')

        # Creamos el QLabel que contendrá la imagen y definimos su tamaño.
        self.label = QLabel(self)
        self.label.setGeometry(50, 50, 100, 100)

        # Ruta al archivo de la imagen y carga de los píxeles.
        ruta_imagen = os.path.join('img', 'python.jpg')
        pixeles = QPixmap(ruta_imagen)

        # Agregamos los píxeles al QLabel.
        self.label.setPixmap(pixeles)

        # Ajustamos el contenido al tamaño del elemento (100 x 100).
        self.label.setScaledContents(True)

        self.show()
```



★ Si la imagen no aparece

`QPixmap` **no levanta un error** si el archivo no existe: simplemente queda vacío y no se ve nada. Revisa la ruta y que tu terminal esté en el directorio correcto (por eso `os.path.join('img', 'python.jpg')` es relativa a donde ejecutas).

4.5 Botones

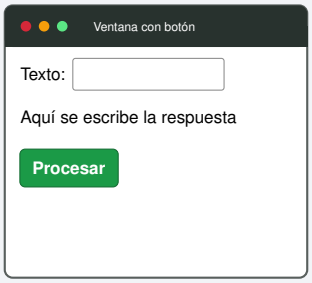
El *widget* de control más básico es el botón, `QPushButton`, que recibe un texto inicial y su *parent* (archivo `3_pushbutton.py`):

```
# Podemos agrupar conjuntos de widgets en alguna estructura.
self.labels = {}
self.labels['label1'] = QLabel('Texto:', self)
self.labels['label1'].move(10, 15)
self.labels['label2'] = QLabel('Aquí se escribe la respuesta', self)
self.labels['label2'].move(10, 50)

self.edit1 = QLineEdit('', self)
self.edit1.setGeometry(45, 15, 100, 20)

# '&' destaca la primera letra (depende de la plataforma).
# sizeHint() entrega un tamaño sugerido para el botón.
self.boton1 = QPushButton('&Procesar', self)
self.boton1.resize(self.boton1.sizeHint())
self.boton1.move(5, 70)
```

Qué verás



Un botón "Procesar" debajo de las etiquetas. Por ahora **no hace nada**: conectarlo a una acción es el tema de la Sección 5.

Detalles útiles

- `sizeHint()` devuelve el tamaño que el *widget* **sugiere** para verse bien; `resize(...)` se lo aplica.
- El carácter `&` al inicio del texto destaca la primera letra (un atajo). Cómo se ve depende de la plataforma.
- Se pueden agrupar *widgets* en estructuras de Python, por ejemplo un diccionario, para accederlos por nombre.

4.6 *Layouts*

Hasta ahora ubicamos todo con **posicionamiento absoluto** (`setGeometry` y `move`). Funciona, pero tiene límites:

- Si cambias el **tamaño de la ventana**, los *widgets* mantienen su tamaño y posición (pruébalo arrastrando el borde con el *mouse*).
- La aplicación se puede ver **distinta** en otras plataformas o configuraciones de pantalla.

Los ***layouts*** resuelven esto: el *layout* se encarga de repartir el espacio y de reacomodar los *widgets* cuando la ventana cambia.

Layout	Cómo ordena los *widgets*
`QHBoxLayout`	En **fila**, de izquierda a derecha (horizontal).
`QVBoxLayout`	En **columna**, de arriba hacia abajo (vertical).
`QGridLayout`	En una **grilla** de filas y columnas.

Los *widgets* se agregan con `addWidget(widget)`, y el *layout* completo se asigna a la ventana con `self.setLayout(...)`. Como un *layout* puede contener a otros, se pueden **anidar**: por ejemplo, un `QHBoxLayout` dentro de un `QVBoxLayout` (con `addLayout`). Con `addStretch(n)` se agregan **espaciadores elásticos** que absorben el espacio sobrante.

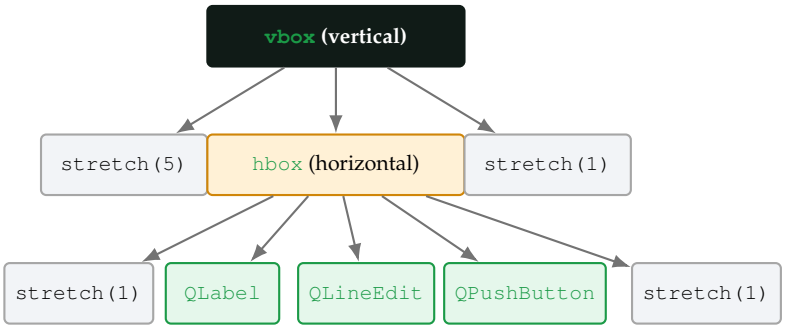
Ejemplo (archivo `4_layout.py`): tres *widgets* en una fila, dentro de una columna con espaciadores.

```python
class MiVentana(QWidget):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.init_gui()
        self.show()

    def init_gui(self) -> None:
        self.setGeometry(100, 100, 300, 300)
        self.label1 = QLabel('Texto:', self)
        self.edit1 = QLineEdit('', self)
        self.edit1.resize(100, 20)
        self.boton1 = QPushButton('&Calcular', self)
        self.boton1.resize(self.boton1.sizeHint())

        # Layout horizontal: widgets con addWidget(), espaciadores con addStretch().
        hbox = QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(self.label1)
        hbox.addWidget(self.edit1)
        hbox.addWidget(self.boton1)
        hbox.addStretch(1)

        # Layout vertical que contiene al horizontal.
        # Ojo con la proporción 5 : 1 de los espaciadores.
        vbox = QVBoxLayout()
        vbox.addStretch(5)
        vbox.addLayout(hbox)
        vbox.addStretch(1)
        self.setLayout(vbox)
```





¿Qué hace este código? El espacio sobrante en vertical se reparte **5 a 1** entre el espaciador de arriba y el de abajo, así que la fila queda **cerca del borde inferior**. En horizontal, los dos `addStretch(1)` reparten el sobrante en partes iguales: la fila queda **centrada**. Al redimensionar la ventana, el *layout* reacomoda todo solo.

★ Para pegar la fila a un costado

Con los dos `addStretch(1)` laterales la fila queda centrada. Si la quieres pegada a la **derecha**, deja solo el espaciador de la **izquierda** (y quita el de la derecha); para la izquierda, al revés. Además, cuando un *layout* manda, los `resize`/`move` manuales de sus *widgets* dejan de tener efecto: el *layout* decide los tamaños.

4.6.1 Grilla: `QGridLayout`

`QGridLayout` divide el espacio en **filas y columnas**; cada *widget* se agrega a una casilla con `addWidget(widget, fila, columna)`. Ideal para algo como el teclado de un celular (archivo `5_gridlayout.py`):

```python
class MiVentana(QWidget):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.init_gui()
        self.show()

    def init_gui(self) -> None:
        # Creamos la grilla para ubicar los widgets de manera matricial.
        grilla = QGridLayout()

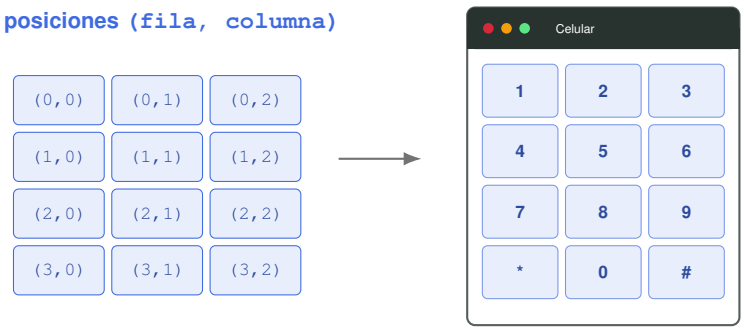
        valores = ['1', '2', '3',
```

```
                  '4', '5', '6',
                  '7', '8', '9',
                  '*', '0', '#']

        # Posiciones (fila, columna) de cada botón en la grilla.
        posiciones = [(i, j) for i in range(4) for j in range(3)]

        for i in range(len(posiciones)):
            posicion = posiciones[i]
            valor = valores[i]
            boton = QPushButton(valor, self)
            grilla.addWidget(boton, *posicion)

        self.setLayout(grilla)
        self.move(300, 150)
        self.setWindowTitle('Celular')
```



Los `(i, j)` se generan con una **lista por comprensión**: `[(i, j) for i in range(4) for j in range(3)]`. Se recorre con `*posicion` para entregar fila y columna a `addWidget`. Al agrandar la ventana los botones se estiran con ella.

Layout	Úsalo cuando...	Agregar elementos
`QHBoxLayout`	quieres una **fila** de *widgets*.	`addWidget(w)`, `addStretch(n)`
`QVBoxLayout`	quieres una **columna**, o apilar otros *layouts*.	`addWidget(w)`, `addLayout(l)`, `addStretch(n)`
`QGridLayout`	los elementos forman una **matriz**.	`addWidget(w, fila, col)`

4.7 POO aplicada a los *widgets*

Toda ventana, botón o etiqueta es un **objeto**, así que podemos usar **herencia** y **atributos propios**. Esto permite personalizar *widgets* y, sobre todo, **reutilizar código**.

4.7.1 Un botón que cuenta sus clics

`MiBoton` hereda de `QPushButton` y agrega un nombre, una posición y un contador (archivo `6_mi-boton.py`):

```
class MiBoton(QPushButton):
    # Recibe dos argumentos extra, además de los regulares de QPushButton:
    # - Un nombre para identificar el botón.
    # - Una posición para ubicarse en la ventana.
    def __init__(self, nombre: str, pos: tuple, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)   # constructor de la clase madre
        self.nombre = nombre
        self.contador = 0                   # contador de instancia

        # Fija su propia geometría
        self.resize(self.sizeHint())
        self.move(*pos)

        # Conecta un clic con el método contar (lo entenderemos en la Sección 5).
        self.clicked.connect(self.contar)

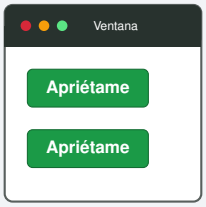
    # Comportamiento propio: aumenta el contador en cada clic.
    def contar(self) -> None:
        self.contador += 1
        print(f"{self.nombre} apretado {self.contador} veces.")


class MiVentana(QWidget):
    def __init__(self, *args, **kwargs) -> None:
```

```
        super().__init__(*args, **kwargs)
        self.init_gui()
        self.show()

    def init_gui(self) -> None:
        self.setGeometry(200, 200, 120, 120)
        self.setMaximumHeight(120)
        self.setMaximumWidth(120)
        # Dos botones de nuestra clase, con atributos extra: nombre y posición.
        self.boton_1 = MiBoton("Botón 1", (10, 20), "Apriétame", self)
        self.boton_2 = MiBoton("Botón 2", (10, 60), "Apriétame", self)
```

Qué verás



Dos botones iguales. Cada clic imprime en la consola, por ejemplo, `Botón 1 apretado 3 veces.` El conteo es **independiente** para cada botón, porque `contador` es un atributo de **instancia**.

4.7.2 Un formulario reutilizando código

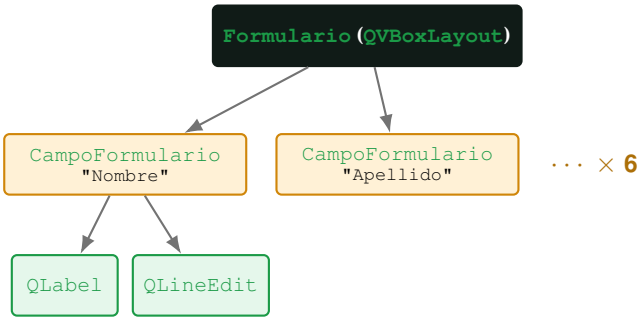
Un formulario es muy regular: cada campo es una etiqueta más un cuadro de texto. Si cada campo son 6 líneas y quieres 6 campos, serían 36 líneas de pura repetición. En cambio, se escribe **una clase para un campo** y se usa varias veces (archivo `7_mi-formulario.py`):

```
class CampoFormulario(QHBoxLayout):
    # Heredamos de un layout horizontal para colocar cada campo.
    def __init__(self, texto: str, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        label = QLabel(f"{texto}: ")
        campo = QLineEdit("")
        self.addStretch(1)
        self.addWidget(label)
        self.addWidget(campo)
        self.addStretch(1)


class Formulario(QWidget):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.setWindowTitle("Formulario")
        self.setGeometry(200, 200, 400, 400)

        contenedor = QVBoxLayout()        # contenedor vertical de campos
        contenedor.addLayout(CampoFormulario("Nombre"))
        contenedor.addLayout(CampoFormulario("Apellido"))
        contenedor.addLayout(CampoFormulario("Dirección"))
        contenedor.addLayout(CampoFormulario("Correo"))
        contenedor.addLayout(CampoFormulario("Usuario"))
        contenedor.addLayout(CampoFormulario("Contraseña"))
        self.setLayout(contenedor)
        self.show()
```





La misma clase se instancia 6 veces: **un cambio** en `CampoFormulario` cambia **todos** los campos.

★ ¿No contradice lo del *parent*?

Aquí `QLabel` y `QLineEdit` se crean **sin** `self` y sí aparecen. Es porque, al hacer `setLayout(contenedor)` en la ventana, el *layout* **adopta** a todos los *widgets* que contiene como hijos de esa ventana. Si no los metes en ningún *layout*, sí necesitas indicar el padre.

✓ Resumen express: *widgets* y *layouts*

- Coordenadas: origen arriba a la izquierda, **Y crece hacia abajo**. `setGeometry(x, y, ancho, alto)` y `move(x, y)`; la posición de un hijo es relativa a su padre.
- `QLabel(texto, padre)`, `QLineEdit(texto, padre)`, `QPushButton(texto, padre)`; imágenes con `QPixmap` dentro de un `QLabel`.
- Los *layouts* (`QHBoxLayout`, `QVBoxLayout`, `QGridLayout`) reacomodan los *widgets* al cambiar el tamaño; se anidan con `addLayout`; `addStretch` reparte espacio sobrante.
- Se hereda de las clases de PyQt para agregar atributos y comportamiento, y reutilizar código.

5 Eventos y señales

Una aplicación PyQt empieza a detectar eventos recién cuando entra a su *main loop*, es decir, después de llamar a `exec()` de `QApplication`. Desde ahí, cada acción del usuario (o de otra parte del programa) se procesa con el elemento apropiado.

5.1 El modelo: fuente, evento y destino

■ Los tres elementos del modelo

- **La fuente del evento**: el objeto que genera el cambio de estado (por ejemplo, un botón).
- **El objeto evento**: encapsula el cambio de estado.
- **El objeto destino**: el objeto al que se quiere notificar del cambio.

La fuente **delega** el manejo del evento al destino pasándole el objeto evento.

PyQt implementa este modelo con el mecanismo de ***signal* y *slot***: cuando ocurre un evento, el objeto activado **emite una señal** (*signal*) hacia una **ranura** (*slot*), y un *slot* puede ser **cualquier elemento llamable** (*callable*) de Python: una función o un método.




```
boton.clicked.connect(self.boton_clickeado)
```
"cuando este botón haga clic, ejecuta este método"

★ Analogía del timbre

La señal es el **timbre** de una casa y el *slot* es quien **abre la puerta**. `connect` es el **cable** que une el timbre con la persona. El botón no sabe quién atenderá: solo toca el timbre.

5.2 Conectar un botón a un *slot*

`connect()` recibe una función o método. **Importante: sin paréntesis.**

Código	Qué ocurre
✓ `boton.clicked.connect(self.metodo)`	Entregamos **el objeto método**; PyQt lo llamará en cada clic.
✗ `boton.clicked.connect(self.metodo())`	Se **ejecuta el método ya** y se entrega su **resultado** (normalmente `None`), no la función; la conexión falla o no hace lo esperado.

El siguiente ejemplo extiende el teclado de celular: cada botón de la grilla se conecta al **mismo** *slot*, que muestra cuál fue el último presionado (archivo `3-eventos-y-señales/1_manejar-eventos.py`).

```python
class MiVentana(QWidget):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)
        self.init_gui()

    def init_gui(self) -> None:
        # Labels para indicar el último botón seleccionado.
        self.label1 = QLabel('Último botón seleccionado:', self)
        self.label2 = QLabel('', self)
        self.grilla = QGridLayout()

        valores = ['1', '2', '3', '4', '5', '6',
                   '7', '8', '9', '*', '0', '#']
        posiciones = [(i, j) for i in range(4) for j in range(3)]

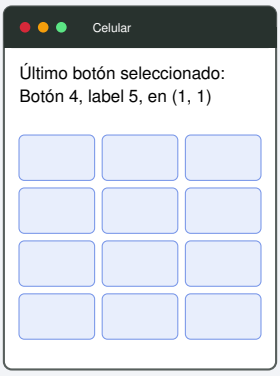
        for i in range(len(valores)):
            boton = QPushButton(valores[i], self)
            # Conectamos clicked de cada botón con el slot. Todos usan el mismo.
            boton.clicked.connect(self.boton_clickeado)
            self.grilla.addWidget(boton, *posiciones[i])

        vbox = QVBoxLayout()
        vbox.addWidget(self.label1)
        vbox.addWidget(self.label2)
        vbox.addStretch(1)
        vbox.addLayout(self.grilla)
        self.setLayout(vbox)

        self.setGeometry(300, 150, 200, 200)
        self.setWindowTitle('Celular')

    def boton_clickeado(self) -> None:
        # sender() retorna el objeto que emitió la señal (un QPushButton).
        boton = self.sender()
        label_boton = boton.text()                    # texto del botón
        idx = self.grilla.indexOf(boton)              # identificador en la grilla
        posicion = self.grilla.getItemPosition(idx)   # (fila, columna, ...)
        self.label2.setText(f'Botón {idx} con label {label_boton}, '
                            f'en fila/columna: {posicion[:2]}.')
```

Qué verás



Al presionar el `5`, la segunda etiqueta se actualiza con su texto, su índice y su posición en la grilla.

¿Qué hace este código? Un solo método atiende a **doce** botones. Para saber *cuál* lo activó usa `sender()`, y con el botón en mano obtiene su texto y su posición en la grilla.

Si `setText` no se refleja

En algunos sistemas operativos, `label.setText(...)` no se refleja de inmediato. Si pasa, llama también a `label.repaint()` para forzar la actualización.

5.3 Obtener al emisor: `sender()`

En ocasiones es necesario saber **cuál objeto envió la señal**. `self.sender()` retorna ese objeto, y con él podemos leer sus atributos (por ejemplo `.text()`). Solo tiene sentido **dentro de un *slot*** que fue llamado por una señal.

En este ejemplo (archivo `2_obtener-sender.py`) hay tres botones: dos usan el mismo *slot* y el tercero cierra la aplicación conectándose directamente a `QCoreApplication.instance().quit`.

```
from PyQt6.QtCore import QCoreApplication

class MiVentana(QWidget):
    def init_gui(self) -> None:
        self.label_estado = QLabel('Estado: -', self)

        self.boton1 = QPushButton('Botón 1', self)
        self.boton1.clicked.connect(self.boton_clickeado)    # sin paréntesis

        self.boton2 = QPushButton('Botón 2', self)
        self.boton2.clicked.connect(self.boton_clickeado)

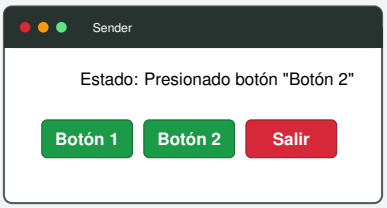
        self.boton3 = QPushButton('Salir', self)
        # instance() entrega la QApplication en ejecución; quit() la cierra.
        self.boton3.clicked.connect(QCoreApplication.instance().quit)

        hbox = QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(self.boton1)
        hbox.addWidget(self.boton2)
        hbox.addWidget(self.boton3)
        hbox.addStretch(1)

        vbox = QVBoxLayout()
        vbox.addStretch(1)
        vbox.addWidget(self.label_estado)
        vbox.addLayout(hbox)
        vbox.addStretch(1)
        self.setLayout(vbox)

    def boton_clickeado(self) -> None:
        sender = self.sender()
        self.label_estado.setText(f'Estado: Presionado botón "{sender.text()}"')
        self.label_estado.resize(self.label_estado.sizeHint())
```

Qué verás



Los dos primeros botones actualizan la etiqueta de estado con su propio texto; `Salir` cierra toda la aplicación.

5.4 Eventos de *mouse*

No solo los botones tienen eventos: **todo *widget*** puede reaccionar al *mouse* y al teclado. `QWidget` incluye métodos como `mousePressEvent`, `mouseReleaseEvent` y `mouseMoveEvent`, que **por defecto no hacen nada**. Para darles comportamiento, basta con **sobrescribirlos** (*override*) en tu propio *widget*.

Señal (ej.: `clicked`)

```
boton.clicked.connect(self.mi_slot)
```

conectas una señal a un método tuyo

Evento de *widget* (ej.: `mousePressEvent`)

```
def mousePressEvent(self, event):  ...
```

sobrescribes un método que ya está conectado

La diferencia: en las señales hay que `connect`-ar; en estos eventos de *widget* la conexión **ya existe**, solo se sobrescribe el método. El objeto `event` (por ejemplo, un `QMouseEvent`) trae los detalles; `event.position().x()` y `event.position().y()` dan las coordenadas dentro del *widget*.

En este ejemplo (archivo `3_mouse-press-y-release-event.py`) hay un `QLabel` de color y se detecta **dónde** se presionó el *mouse*:

```python
class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.setGeometry(100, 100, 110, 400)
        self.label = QLabel("Haz click en mí", self)
        self.label.setGeometry(10, 10, 90, 100)
        self.label.setStyleSheet("background-color: lightblue;")
        self.click_dentro_del_label = False

    def mousePressEvent(self, event: QMouseEvent) -> None:
        x = event.position().x()
        y = event.position().y()
        print(f"El mouse fue presionado en {x},{y}")
        # underMouse() dice si el cursor está sobre ese widget.
        self.click_dentro_del_label = self.label.underMouse()
        if self.click_dentro_del_label:
            print("\tFue presionado dentro del QLabel")
        else:
            print("\tFue presionado fuera del QLabel")

    def mouseReleaseEvent(self, event: QMouseEvent) -> None:
        x = event.position().x()
        y = event.position().y()
        print(f"El mouse fue liberado en {x},{y}")
        if self.click_dentro_del_label:
            print("\tAntes se había presionado dentro del QLabel")
        else:
            print("\tAntes habías presionado fuera del QLabel")
```

¿Qué hace este código? Al presionar, guarda un **estado** (`click_dentro_del_label`); al soltar, lo usa para informar dónde había comenzado el clic. Así se pueden distinguir, por ejemplo, arrastres que empiezan dentro o fuera de un elemento.

5.4.1 Movimiento permanente: *mouse tracking*

Por defecto, `mouseMoveEvent` solo se dispara **mientras hay un botón presionado**. Para recibir la posición del *mouse* **aunque no se presione nada**, se activa el *tracking* con `setMouseTracking(True)` (archivo

`4_mouse-move-event.py`):

```python
class MiVentana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.setGeometry(100, 100, 110, 400)
        self.label = QLabel("Haz click en mí", self)
        self.label.setGeometry(10, 10, 90, 100)
        self.label.setStyleSheet("background-color: lightblue;")

        self.setMouseTracking(True)          # tracking en la ventana
        self.label.setMouseTracking(True)    # y también sobre el label

    def mouseMoveEvent(self, event: QMouseEvent) -> None:
        x = event.position().x()
        y = event.position().y()
        print(f"El mouse se mueve... está en {x},{y}")
```

★ Detalles que ayudan

- Si además quieres seguir el *mouse* cuando está sobre el `QLabel`, también hay que activar el *tracking* en el *label*.
- `event.position()` entrega un `QPointF` (coordenadas decimales). Verás también `event.pos()`, que entrega enteros (`QPoint`); en Qt6 está marcado como obsoleto pero sigue funcionando.
- Hay muchos más: `enterEvent`, `leaveEvent`, `dragEnterEvent`... Consulta la documentación.

5.5 Eventos del teclado

Cuando el usuario **presiona o suelta una tecla**, `QWidget` llama a `keyPressEvent` y `keyReleaseEvent`. Igual que con el *mouse*, por defecto no hacen nada y se sobrescriben. El `QKeyEvent` trae el texto de la tecla (`event.text()`) y su código (`event.key()`). Ejemplo (archivo `5_key-press-y-release-event.py`):

```python
class MiVentana(QWidget):
    def inicializa_gui(self) -> None:
        self.teclas_presionadas = set()
        self.estado = QLabel('No se está presionando una tecla.', self)
        self.estado.move(20, 10)
        self.setGeometry(300, 300, 600, 150)
        self.setWindowTitle('Teclado')
        self.show()

    def keyPressEvent(self, event: QKeyEvent) -> None:
        # Se ejecuta al presionar una tecla.
        nuevo_texto = event.text()
        nuevo_codigo = event.key()
        self.teclas_presionadas.add((nuevo_texto, nuevo_codigo))
        self.estado.setText(f'Presionaron las teclas: {list(self.teclas_presionadas)}')
        self.estado.resize(self.estado.sizeHint())

    def keyReleaseEvent(self, event: QKeyEvent) -> None:
        # Se ejecuta al soltar una tecla.
        texto_eliminado = event.text()
        codigo_eliminado = event.key()
        self.teclas_presionadas.remove((texto_eliminado, codigo_eliminado))
        if len(self.teclas_presionadas) == 0:
            self.estado.setText('No se está presionando una tecla.')
        else:
            self.estado.setText(f'Presionaron las teclas: {list(self.teclas_presionadas)}')
        self.estado.resize(self.estado.sizeHint())
```

¿Qué hace este código? Mantiene un `set` con las teclas que están **actualmente apretadas**: agrega al presionar y quita al soltar. Así puede mostrar combinaciones (varias teclas a la vez).

! Cuidado con `remove`

Si sueltas una tecla cuyo texto cambió desde que la presionaste (por ejemplo, presionas `Shift` + `a`, escribes `A`, y sueltas `Shift` antes que la `a`), el par `(texto, codigo)` que se intenta quitar puede no estar en el `set` y `remove` levanta **`KeyError`**. Una variante más robusta es usar `discard`, que no falla si el elemento no existe.

5.6 Señales personalizadas

Además de las señales de PyQt (como `clicked`), puedes **definir las tuyas** con `pyqtSignal`. Se usan tres piezas:

Pieza	Rol
`pyqtSignal()`	Define la señal. Se escribe como **atributo de clase** en un objeto que herede de `QObject`.
`señal.emit()`	**Emite** la señal: avisa que ocurrió el evento.
`señal.connect(f)`	Define **qué función o método** se ejecuta cuando se emite.

Ejemplo (archivo `6_crear-pyqtsignals.py`): dos ventanas separadas; al presionar la izquierda se edita el texto de la derecha.

```python
class VentanaPresionable(QWidget):
    # La señal es un atributo de CLASE.
    senal_escribir = pyqtSignal()

    def __init__(self) -> None:
        super().__init__()
        self.inicializa_gui()

    def inicializa_gui(self) -> None:
        self.etiqueta = QLabel('Presiona esta ventana', self)
        self.etiqueta.move(20, 10)
        self.setGeometry(300, 300, 290, 150)
        self.setWindowTitle('Emite señal')
        self.show()

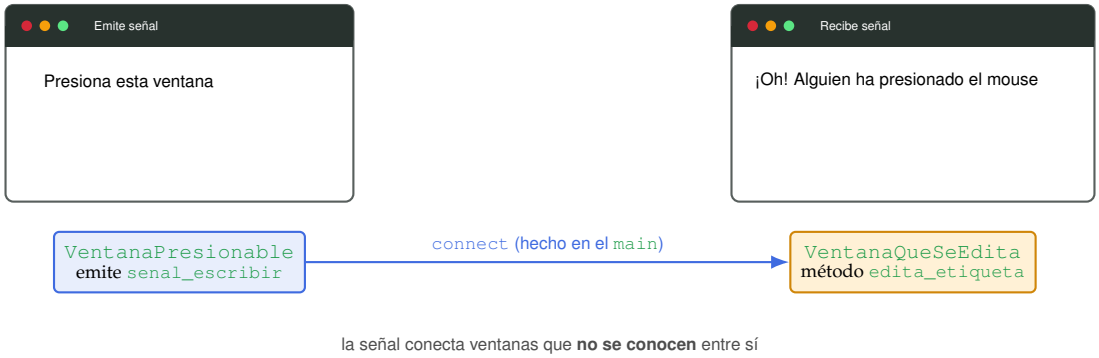
    def mousePressEvent(self, event: QMouseEvent) -> None:
        # Emitimos la señal: los métodos conectados se llaman automáticamente.
        self.senal_escribir.emit()


class VentanaQueSeEdita(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.inicializa_gui()

    def inicializa_gui(self) -> None:
        self.etiqueta = QLabel('', self)
        self.etiqueta.move(20, 10)
        self.setGeometry(700, 300, 290, 150)
        self.setWindowTitle('Recibe señal')
        self.show()

    # Este es el método que conectaremos a la señal.
    def edita_etiqueta(self) -> None:
        self.etiqueta.setText('¡Oh! Alguien ha presionado el mouse')
        self.etiqueta.resize(self.etiqueta.sizeHint())


if __name__ == '__main__':
    app = QApplication([])
    ventana_click = VentanaPresionable()
    ventana_edit = VentanaQueSeEdita()
    # Conectamos la señal con el método que debe activar.
    ventana_click.senal_escribir.connect(ventana_edit.edita_etiqueta)
    sys.exit(app.exec())
```



¿Quién hace qué? `VentanaPresionable` **emite**; `VentanaQueSeEdita` **recibe** y edita su etiqueta; y el `main` es quien las **conecta**. Ninguna de las dos clases conoce a la otra.

5.6.1 (Abre paréntesis...) Preguntas frecuentes sobre las señales

¿Por qué la señal vive dentro de una clase y no suelta?

Las `pyqtSignal` deben estar encapsuladas por objetos que **hereden de `QObject`**. Fuera de una clase se pueden instanciar, pero no se podrán conectar ni emitir. `VentanaPresionable` sirve porque hereda de `QWidget`, y todo `QWidget` es un `QObject`.

¿Por qué es atributo de clase y no de instancia?

Es una exigencia del diseño interno de PyQt: si la defines como atributo de instancia, no se podrá conectar ni emitir. Pruébalo y verás que falla.

¿Todas las instancias comparten la misma señal?

No, aunque se defina como atributo de clase. Por la implementación interna de PyQt, **cada instancia tiene su propia señal**, distinta de la de las demás.

! Solo ASCII en el nombre

El nombre del atributo que contiene la señal solo puede usar caracteres **ASCII**: nada de `ñ` ni tildes. Por eso verás `senal_escribir` y no `señal_escribir`.

5.7 Señales que llevan información

Una señal puede **transportar datos**: al crear la `pyqtSignal` se indican los **tipos** de lo que enviará con `emit`, y el método conectado los recibe como **argumentos**. El siguiente ejemplo (archivo `7_enviar-informacion-pyqtsignals.py`) tiene tres señales.

```
class VentanaPresionable(QWidget):
    senal_simple = pyqtSignal()                 # sin datos
    senal_texto = pyqtSignal(str)               # envía un str
    senal_coordenadas = pyqtSignal(int, int)    # envía dos ints

    # ... (inicializa_gui crea un QLabel y un QLineEdit, y activa setMouseTracking)

    def mousePressEvent(self, event: QMouseEvent) -> None:
        self.senal_simple.emit()
        # Enviamos el texto que contiene el QLineEdit.
        self.senal_texto.emit(self.etiqueta.text())

    def mouseMoveEvent(self, event: QMouseEvent) -> None:
        # Enviamos la posición del mouse.
        self.senal_coordenadas.emit(event.pos().x(), event.pos().y())


class VentanaQueSeEdita(QWidget):
    def edita_etiqueta_click(self) -> None:             # sin argumentos
        self.etiqueta_1.setText('¡Oh! Alguien ha presionado el mouse')

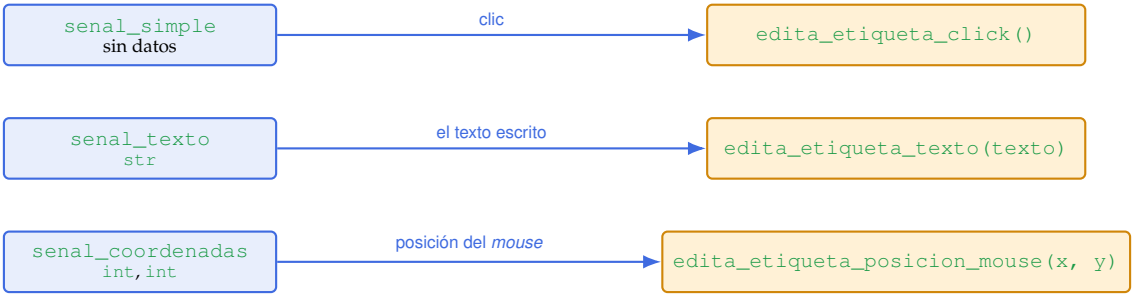
    def edita_etiqueta_texto(self, texto) -> None:      # recibe un str
```

```
        self.etiqueta_2.setText(f'Recibí del evento: {texto}')

    def edita_etiqueta_posicion_mouse(self, x, y) -> None:   # recibe dos ints
        self.etiqueta_3.setText(f'Recibí posiciones: {x}, {y}')


if __name__ == '__main__':
    app = QApplication([])
    ventana_click = VentanaPresionable()
    ventana_edit = VentanaQueSeEdita()
    ventana_click.senal_simple.connect(ventana_edit.edita_etiqueta_click)
    ventana_click.senal_texto.connect(ventana_edit.edita_etiqueta_texto)
    ventana_click.senal_coordenadas.connect(ventana_edit.edita_etiqueta_posicion_mouse)
    sys.exit(app.exec())
```

Definición	Emisión	*Slot* conectado
`pyqtSignal()`	`senal.emit()`	def slot(self)
`pyqtSignal(str)`	`senal.emit("hola")`	def slot(self, texto)
`pyqtSignal(int, int)`	`senal.emit(10, 20)`	def slot(self, x, y)



★ Datos más complejos

Los tipos pueden ser cualquiera que necesites: también se pueden enviar estructuras como un `dict` (con `pyqtSignal(dict)`). La regla es que los tipos declarados coincidan con lo que se `emit`e y con lo que el *slot* espera recibir.

✓ Resumen express: eventos y señales

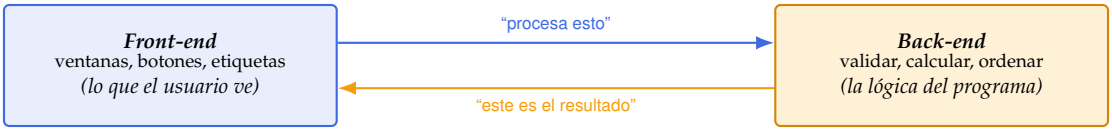
- Modelo: **fuente** → **evento** → **destino**. En PyQt: la fuente *emite una señal* y el *slot* (cualquier *callable*) la recibe. Se unen con `connect` (**sin paréntesis**).
- `self.sender()` dice quién emitió la señal (solo dentro de un *slot*).
- Eventos de *widget* (`mousePressEvent`, `mouseMoveEvent`, `keyPressEvent`...): se **sobrescriben**. El movimiento sin clic requiere `setMouseTracking(True)`.
- Señales propias: `pyqtSignal` como **atributo de clase** de un `QObject`, nombre solo ASCII; se emiten con `emit` y se conectan con `connect`.
- Las señales pueden **transportar datos**: los tipos se declaran en `pyqtSignal(...)` y llegan como argumentos al *slot*.

6 Diseño de software: *front-end* y *back-end*

6.1 Dos capas, dos responsabilidades

En ingeniería de *software*, ***front-end*** y ***back-end*** nombran la separación entre la **capa de presentación** y la **capa de datos y lógica**. En una interfaz gráfica:

- el ***front-end*** es la **interfaz gráfica** con la que el usuario interactúa (ventanas, botones, etiquetas);
- el ***back-end*** es la **lógica** que hay detrás (validar, calcular, ordenar, guardar).



Esta separación responde a un principio de diseño de calidad: buscar **alta cohesión y bajo acoplamiento**.

Cohesión y acoplamiento

Cohesión: cada componente debe hacer **solo las tareas para las que fue creada**, delegando el resto a otras. Ejemplo: una clase `SimulacionParqueAtracciones` cohesiva tendría métodos como `iniciar_simulacion()` o `detener_simulacion()`, pero no `limpiar_atraccion()` ni `ingresar_clientes_a_restaurant()`, que corresponden a otras clases.
Acoplamiento: ocurre cuando modificar una componente **obliga a modificar otra** para que el cambio sea correcto y completo. Si al cambiar los atributos de la clase `A` también debes cambiar los de la clase `B`, hay *alto acoplamiento*. Un buen diseño lo reduce.

ALTA COHESIÓN

cada pieza hace **una sola cosa** y la hace bien

BAJO ACOPLAMIENTO

cambiar una pieza **no obliga a cambiar** las demás

6.2 ¿Por qué separar? Las ventajas

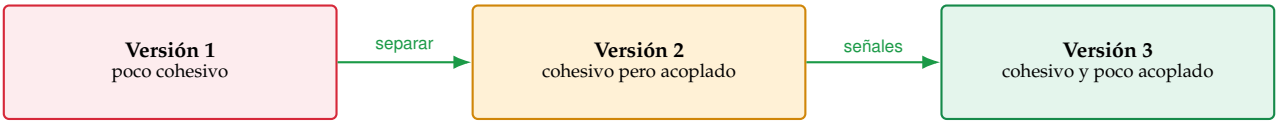
Ventaja	En qué consiste
Modularidad	Puedes cambiar una parte sin afectar la otra. Editas el *front-end* suponiendo que las funciones del *back-end* mantienen su comportamiento, y puedes reescribir el *back-end* (más eficiente, en varios archivos) con un solo punto de conexión con el *front-end*.
Uso de recursos	A veces el *front-end* corre en un computador y el *back-end* en otro. Los cálculos costosos no recargan la interfaz. Ejemplo: en un navegador, los datos se calculan en un servidor y tu computador solo los muestra.
Escalamiento	El *software* puede crecer sin interferir con lo antiguo, y el procesamiento se puede distribuir en múltiples réplicas del *back-end* (muy usado en la web).
Experticia	Quienes desarrollan interfaces y quienes desarrollan lógica suelen tener experiencia distinta: separarlos permite lo mejor de cada uno.
Mantención	Puedes hacer *testing* parte por parte y corregir sin propagar cambios a muchos lugares del código.
Evolución y versionamiento	Puedes cambiar completamente una parte mientras se mantenga **la misma interfaz de métodos**. Un *back-end* hecho para PyQt5 sirve para PyQt6, o incluso para un *back-end* web.

★ ¿Y el costo?

Estas ventajas pueden convertirse en costos si hay que mantener **dos equipos** o **dos programas**. Aun así, las ventajas siguen siendo mayores.

6.3 Ejemplo: de un programa enredado a uno bien diseñado

Construiremos el mismo programa tres veces. Recibe una **lista de números separados por coma** y muestra la lista **ordenada**.



6.3.1 Versión 1: programa poco cohesivo

Todo ocurre dentro de una sola clase `Ventana` (archivo `1_baja-cohesion/main.py`). Al hacer clic se: (1) quitan espacios y comas sobrantes; (2) valida que el texto sea válido, y si no, avisa en una etiqueta; (3) ordena y muestra el resultado.

```
class Ventana(QWidget):
    def __init__(self) -> None:
        super().__init__()
        self.inicializa_gui()

    def inicializa_gui(self) -> None:
        self.etiqueta = QLabel('Ingresa una lista de números '
                               'separados por comas:', self)
        self.etiqueta.move(20, 10)
        self.etiqueta.resize(self.etiqueta.sizeHint())

        self.input = QLineEdit('', self)
        self.input.setGeometry(20, 40, 360, 20)

        self.boton = QPushButton('Ordenar', self)
        self.boton.setGeometry(20, 70, 360, 30)
        self.boton.clicked.connect(self.revisar_y_ordenar)

        self.resultado = QLabel('', self)
        self.resultado.move(20, 100)
        self.resultado.resize(self.resultado.sizeHint())

        self.setGeometry(700, 300, 400, 200)
        self.setWindowTitle('Ordenador de números')
        self.show()

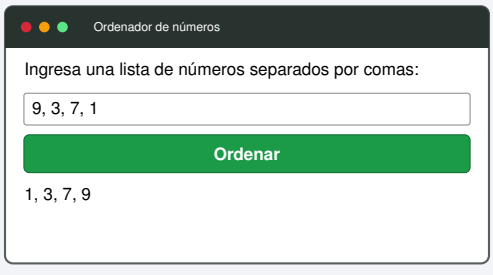
    def revisar_y_ordenar(self) -> None:
        texto_input = self.input.text()
        if not texto_input:
            return

        texto_input = texto_input.replace(' ', '').strip(',')
        if not texto_input.replace(',', '').isnumeric():
            self.resultado.setText('Input no válido')
            self.resultado.resize(self.resultado.sizeHint())
            return

        lista_de_numeros = [int(porcion) for porcion in texto_input.split(',')]
        lista_de_numeros.sort()
        texto_resultado = ", ".join([str(numero) for numero in lista_de_numeros])

        self.resultado.setText(texto_resultado)
        self.resultado.resize(self.resultado.sizeHint())
        self.resultado.repaint()
```

■ Qué verás



Escribes números separados por coma, haces clic en `Ordenar` y aparece la lista ordenada (o "Input no válido").

! Diagnóstico: baja cohesión

El programa funciona, pero la clase `Ventana`, que es un componente **visual**, hace **todo**. El método `revisar_y_ordenar` mezcla **procesamiento de datos** con **tareas visuales**. Las responsabilidades no se reparten entre componentes: eso es código poco cohesivo.

Ventana
- dibuja la interfaz
- lee el texto escrito
- limpia y valida los datos ✗
- ordena la lista ✗
- actualiza las etiquetas

las dos líneas marcadas **no** son tareas visuales: pertenecen al *back-end*.

6.3.2 Versión 2: cohesivo pero acoplado

Ahora separamos en dos módulos: `backend.py` (validar y ordenar) y `frontend.py` (lo visual). Archivos en `2_alta-cohesion-alto-acoplamiento/`.

```python
# ---------- backend.py ----------
def es_valido(texto: str) -> bool:
    for valor in texto.split(','):
        if not valor.isnumeric():
            return False
    return True


def ordenar(lista_de_numeros: list) -> list:
    lista_de_numeros.sort()
    return lista_de_numeros


def procesar_input(texto_input: str) -> str:
    texto_input = texto_input.replace(' ', '').strip(',')
    if not es_valido(texto_input):
        return 'Input no válido'
    lista_de_numeros = [int(porcion) for porcion in texto_input.split(',')]
    numeros_ordenados = ordenar(lista_de_numeros)
    texto_resultado = ", ".join([str(numero) for numero in numeros_ordenados])
    return texto_resultado

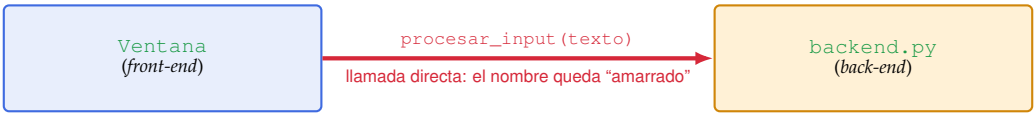

# ---------- frontend.py (solo lo que cambia) ----------
from backend import procesar_input

class Ventana(QWidget):
    # inicializa_gui igual que antes; el botón se conecta a boton_clickeado
    def boton_clickeado(self) -> None:
        texto_input = self.input.text()
        texto_resultado = procesar_input(texto_input)    # llamada DIRECTA al back-end
        self.resultado.setText(texto_resultado)
        self.resultado.resize(self.resultado.sizeHint())
        self.resultado.repaint()
```

Hay dos beneficios: todo el procesamiento queda en un archivo aparte y ordenado, y la ventana se simplifica a tareas visuales. Es decir, ganamos **cohesión**. Además ya califica como arquitectura *front-end / back-end*.

! Diagnóstico: todavía hay acoplamiento

La parte visual **llama directamente** a la lógica (`procesar_input`). Si se cambia el nombre de esa función en `backend.py`, **hay que cambiarlo también** en `Ventana` y en cualquier otro lugar que la use. Con una sola función parece poco grave, pero en un programa grande con muchas funciones importadas se vuelve un problema grande. Hay que llamar al *back-end* de alguna forma… ¿cómo hacerlo **sin llamar a la función directamente**? ¡Con **señales**!



6.3.3 Versión 3: cohesivo y poco acoplado

Se elimina el llamado directo entre componentes y se delega todo a **señales**. Se crean dos:

- `senal_procesar` (en el *front-end*): avisa al *back-end* que hay **input nuevo** por procesar. Lleva un `str`.

- `senal_actualizar` (en el *back-end*): avisa al *front-end* que hay que **actualizar la ventana** con el resultado. Lleva un `str`.

Las funciones de procesamiento se encapsulan en una clase `Procesador` que hereda de `QObject` (para poder contener la señal). Archivos en `3_alta-cohesion-bajo-acoplamiento/`.

```python
# ---------- backend.py ----------
from PyQt6.QtCore import QObject, pyqtSignal


class Procesador(QObject):
    senal_actualizar = pyqtSignal(str)

    def es_valido(self, texto: str) -> bool:
        for valor in texto.split(','):
            if not valor.isnumeric():
                return False
        return True

    def ordenar(self, lista_de_numeros: list) -> list:
        lista_de_numeros.sort()
        return lista_de_numeros

    def procesar_input(self, texto_input: str) -> None:
        texto_input = texto_input.replace(' ', '').strip(',')
        if not self.es_valido(texto_input):
            self.actualizar_interfaz('Input no válido')
            return
        lista_de_numeros = [int(porcion) for porcion in texto_input.split(',')]
        numeros_ordenados = self.ordenar(lista_de_numeros)
        texto_resultado = ", ".join([str(numero) for numero in numeros_ordenados])
        self.actualizar_interfaz(texto_resultado)

    def actualizar_interfaz(self, texto: str) -> None:
        self.senal_actualizar.emit(texto)

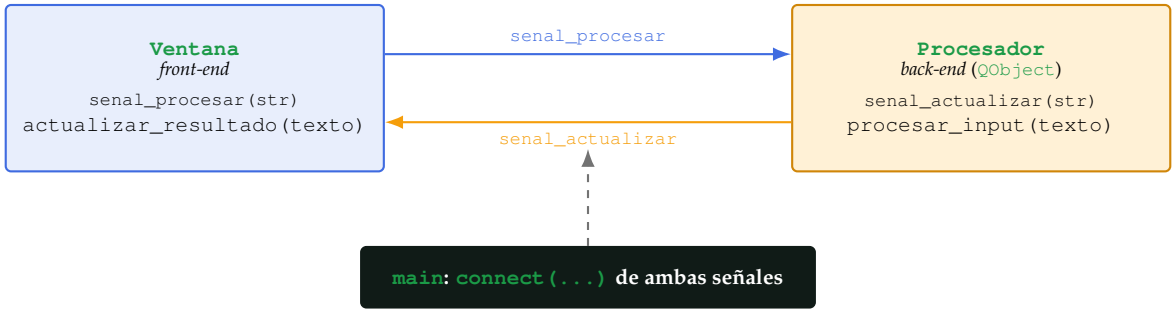

# ---------- frontend.py ----------
from backend import Procesador

class Ventana(QWidget):
    senal_procesar = pyqtSignal(str)

    # inicializa_gui igual que antes
    def boton_clickeado(self) -> None:
        texto_input = self.input.text()
        self.senal_procesar.emit(texto_input)        # solo AVISA, no llama al back-end

    def actualizar_resultado(self, texto: str) -> None:
        self.resultado.setText(texto)
        self.resultado.resize(self.resultado.sizeHint())
        self.resultado.repaint()


if __name__ == '__main__':
    app = QApplication([])
    procesador = Procesador()
    ventana = Ventana()
    # El ÚNICO punto de acoplamiento: conectar las señales con los métodos.
    procesador.senal_actualizar.connect(ventana.actualizar_resultado)
    ventana.senal_procesar.connect(procesador.procesar_input)
    sys.exit(app.exec())
```



ninguna de las dos clases llama a la otra: solo **emiten** y **reciben**

★ ¿Por qué esto reduce el acoplamiento?

Las clases `Ventana` y `Procesador` **nunca llaman código de la otra**: solo ejecutan código propio y delegan el envío de información a las señales. Si cambias el nombre de `procesar_input`, solo se toca la línea del `connect` en el `main`. Es imposible lograr acoplamiento cero (la conexión misma es un acoplamiento), pero sí **minimizarlo**.

6.3.4 Comparación de las tres versiones

	V1	V2	V3
Estructura	Una sola clase lo hace todo.	`frontend.py` + `backend.py` (funciones).	`Ventana` + `Procesador` (clases con señales).
Cohesión	✗ Baja.	✓ Alta.	✓ Alta.
Acoplamiento	— (todo junto)	✗ Alto: el *front* llama al *back*.	✓ Bajo: solo el `connect` del `main`.
¿Quién llama a quién?	Nadie: es un bloque.	*Front* → *back*, directo.	Se comunican por señales.

6.4 Preguntas frecuentes de este ejemplo

■ ¿Por qué `Procesador` hereda de `QObject`?

Por una regla de PyQt: **toda clase que define una `pyqtSignal` como atributo debe heredar de `QObject`** y llamar a su constructor (`super().__init__()`). `Ventana` no necesita hacerlo explícitamente porque ya es un `QWidget`, que es un `QObject`.

■ ¿Por qué una señal se crea en el *back-end* y otra en el *front-end*?

Fue una **decisión de diseño** del ejemplo: cada señal se crea en quien la **emite**. Se puede definir de otra forma, siempre que las conexiones se hagan en el orden correspondiente.

■ ¿No es demasiado complicado para algo tan simple?

Quizás un poco, pero eso se nota porque el programa es pequeño. En programas más extensos esta separación es mucho más evidente y provechosa.

★ Una nota sobre la validación

Ojo con el validador del ejemplo: `str.isnumeric()` solo acepta dígitos, así que `'-3'` o `'2.5'` se rechazan como "Input no válido". Si quieres soportar negativos o decimales, solo hay que cambiar el *back-end*: el *front-end* ni se entera, justamente por la separación.

✓ Resumen express: *front-end* y *back-end*

- *Front-end* = interfaz; *back-end* = lógica. Buscamos **alta cohesión** (cada pieza una responsabilidad) y **bajo acoplamiento** (cambiar una pieza no obliga a cambiar otra).
- Para **alta cohesión**: separar responsabilidades en módulos y clases con objetivos claros.

- Para **bajo acoplamiento**: independizar los componentes; las **señales** permiten comunicarlos sin llamarse directamente.
- Una clase que define una `pyqtSignal` debe heredar de `QObject`; las conexiones se hacen en un solo lugar (el `main`).

7 Conexión entre múltiples ventanas

Al crear interfaces con varias ventanas es natural querer comunicarlas y, en particular, **cambiar de una a otra**. Desde los primeros ejemplos uno se siente tentado a usar `show()` y `hide()`. Veremos tres intentos, por qué uno falla, y cuál es la mejor solución.

7.1 Intento 1: crear la ventana dentro del método (falla)

Una ventana instancia otra e intenta abrirla (archivo `1_no-abre-otra-ventana.py`):

```python
class Ventana(QWidget):
    def __init__(self, titulo: str, x: int, y: int) -> None:
        super().__init__()
        self.setWindowTitle(titulo)
        self.setGeometry(x, y, 200, 50)
        self.boton = QPushButton("Abrir otra ventana", self)
        self.boton.clicked.connect(self.abrir_otra_ventana)
        self.show()

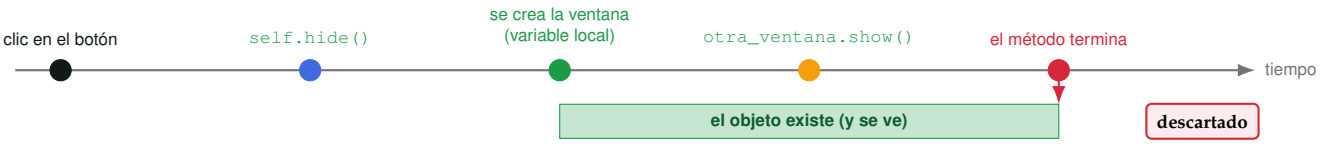
    def abrir_otra_ventana(self) -> None:
        self.hide()  # Esconder la ventana actual
        otra_ventana = Ventana("Otra ventana", 300, 100)  # Crear otra
        otra_ventana.show()  # Mostrar nueva ventana
```

! Resultado inesperado: no se muestra la segunda ventana

La primera ventana se esconde, pero **la segunda nunca se ve**.

7.1.1 ¿Qué ocurrió? Variables locales y objetos descartados

Dentro de `abrir_otra_ventana`, `otra_ventana` es una **variable local** del método. Cuando el método termina, Python **descarta** esa variable, y con ella el *widget* recién creado (y mostrado): ya no queda nadie que lo referencie.



al terminar el método, Python elimina la variable local ⇒ se destruye el *widget*

7.2 Intento 2: crear la otra ventana antes y entregarla

Si la segunda ventana existe **fuera** del método, el término del método no la destruye. Creamos la segunda primero y se la pasamos como argumento a la inicial (archivo `2_no-abre-ventana-inicial.py`):

```python
class Ventana(QWidget):
    def __init__(self, titulo: str, x: int, y: int, otra_ventana: QWidget | None = None) -> None:
        super().__init__()
        self.otra_ventana = otra_ventana
        self.setWindowTitle(titulo)
        self.setGeometry(x, y, 200, 50)
        self.boton = QPushButton("Abrir otra ventana", self)
        self.boton.clicked.connect(self.abrir_otra_ventana)

    def abrir_otra_ventana(self) -> None:
```

```
        if self.otra_ventana is not None:
            self.hide()                 # Esconder la ventana actual
            self.otra_ventana.show()    # Mostrar otra ventana


if __name__ == '__main__':
    app = QApplication([])
    # La segunda ventana se crea antes, de forma independiente.
    otra_ventana = Ventana("Otra ventana", 300, 100)
    # La ventana inicial recibe como argumento a otra_ventana.
    ventana = Ventana("Inicial", 100, 100, otra_ventana)
    ventana.show()
    sys.exit(app.exec())
```

Esto **sí funciona**, porque la variable `otra_ventana` vive en el `main` y además queda guardada como atributo de la primera ventana.

7.3 Intento 3: guardar la ventana como atributo de instancia

Siguiendo la misma idea, guardar la nueva ventana como **atributo de instancia** también arregla el problema (archivo `3_abre-otras-ventanas-infinitas.py`):

```
    def abrir_otra_ventana(self) -> None:
        self.hide()
        self.otra_ventana = Ventana("Otra ventana", 300, 100)    # atributo de instancia
        self.otra_ventana.show()
```

Funciona porque, al salir del método, el objeto queda **referenciado en un nivel superior**: la instancia de la primera ventana. (El nombre del archivo avisa del efecto colateral: cada ventana nueva puede abrir otra, *ad infinitum*.)

■ Regla de oro de PyQt

Siempre hay que guardar una referencia de tus objetos PyQt en memoria: como atributo, o dentro de una estructura de datos (por ejemplo, una lista) que a su vez sea un atributo. Si no queda ninguna referencia, Python descarta el objeto y su ventana desaparece.

★ Analogía de la pizarra

Una variable local es un dibujo en una **pizarra que se borra** al terminar la clase. Un atributo (o una variable del `main`) es el dibujo **en tu cuaderno**: sigue ahí cuando termina el método.

7.4 ¿Es esa la mejor manera de modelarlo?

La solución anterior sirve para casos pequeños, pero tiene un problema de diseño: **modela una dependencia**. La segunda ventana solo existe si existe la primera, y eso no vale para todos los casos. Las ventanas pueden ser **independientes** y aun así un evento en una puede hacer aparecer a la otra. En programas grandes, con muchas ventanas que se llaman entre sí, esa dependencia se vuelve difícil de mantener.

7.5 Señales al rescate

Igual que en el *front-end*/*back-end*, las **señales** permiten **desacoplar**: cada ventana es independiente y se conectan por señales (archivo `4_abre-ventanas-con-signal.py`):

```
from PyQt6.QtCore import pyqtSignal


class Ventana(QWidget):
    # Señal para avisar a otra ventana.
    senal_abrir_otra_ventana = pyqtSignal()

    def __init__(self, titulo: str, x: int, y: int) -> None:
        super().__init__()
        self.setWindowTitle(titulo)
        self.setGeometry(x, y, 200, 50)
        # El botón se conecta a un método propio.
        self.boton = QPushButton("Abrir otra ventana", self)
        self.boton.clicked.connect(self.abrir_otra_ventana)

    def abrir_otra_ventana(self) -> None:
```

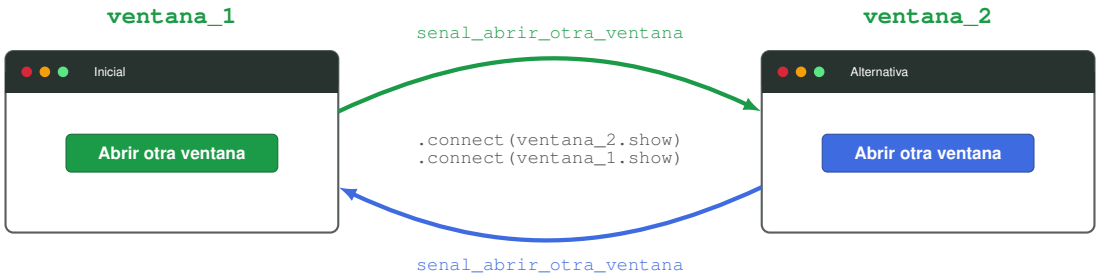
```
        self.hide()
        self.senal_abrir_otra_ventana.emit()


if __name__ == '__main__':
    app = QApplication([])

    # Dos ventanas independientes; cada una con su propia señal.
    ventana_1 = Ventana("Inicial", 100, 100)
    ventana_2 = Ventana("Alternativa", 500, 100)

    ventana_1.senal_abrir_otra_ventana.connect(ventana_2.show)
    ventana_2.senal_abrir_otra_ventana.connect(ventana_1.show)

    ventana_1.show()
    sys.exit(app.exec())
```



cada ventana **no sabe** que la otra existe: la conexión se hace en el `main`

¿Qué hace este código? Al hacer clic, cada ventana se esconde y **emite** su señal. Quien *conectó* esa señal en el `main` (el método `show` de la otra ventana) hace aparecer a la otra. Cada instancia de `Ventana` es independiente: no contiene una instancia de la otra. Además, `ventana_1` y `ventana_2` son variables del `main`, así que **siguen referenciadas** mientras el programa corre.

7.5.1 Comparación de los cuatro intentos

Intento	¿Funciona?	Por qué	Dependencia
1. Variable local	✗ No	La variable se descarta al terminar el método.	—
2. Pasar por argumento	✓ Sí	El objeto vive fuera del método.	Una ventana guarda a la otra.
3. Atributo de instancia	✓ Sí	Queda referenciada por la primera ventana.	La segunda solo existe si existe la primera.
4. Señales	✓ Sí	Cada ventana vive en el `main`; se comunican emitiendo y conectando.	Mínima: solo los `connect`.

✓ Resumen express: varias ventanas

- Una ventana creada en una **variable local** desaparece al terminar el método: **siempre guarda una referencia** (atributo, o estructura que sea atributo).
- Guardar la ventana nueva como atributo funciona, pero crea una **dependencia** entre ventanas.
- La mejor modelación: ventanas **independientes** conectadas por **señales** (`emit` en una, `connect(otra.show)` en el `main`).

8 Hoja de referencia

Herramienta	Qué hace	Detalle clave
`QApplication([])`	Crea la app y su *main loop*.	Una sola, antes de todo *widget*.
`app.exec()`	Inicia el *loop* de eventos.	Retorna un código; `sys.exit(codigo)`.
`QWidget`	Ventana y base de todo *widget*.	`show()`, `hide()`, `setWindowTitle(...)`.
`setGeometry(x, y, w, h)`	Posición y tamaño.	Origen arriba a la izquierda; Y crece hacia abajo.
`move(x, y)` / `resize(w, h)`	Solo posición / solo tamaño.	Relativo al *widget* padre.
`QLabel(texto, padre)`	Etiqueta de texto o imagen.	`setText`, `setPixmap`, `setScaledContents`.
`QLineEdit(texto, padre)`	Cuadro de texto.	`text()` entrega lo escrito.
`QPushButton(texto, padre)`	Botón.	Señal `clicked`; `sizeHint()`.
`QPixmap(ruta)`	Carga una imagen.	No falla si la ruta no existe.
`QHBoxLayout` / `QVBoxLayout`	Fila / columna.	`addWidget`, `addLayout`, `addStretch`.
`QGridLayout`	Grilla.	`addWidget(w, fila, col)`.
`setLayout(layout)`	Asigna el *layout* a la ventana.	Adopta a sus *widgets* como hijos.
`señal.connect(f)`	Une señal con *slot*.	**Sin paréntesis**.
`self.sender()`	Quién emitió la señal.	Solo dentro de un *slot*.
`mousePressEvent(event)`	Clic del *mouse*.	Se sobrescribe; `event.position()`.
`mouseMoveEvent(event)`	Movimiento del *mouse*.	Sin clic: `setMouseTracking(True)`.
`keyPressEvent(event)`	Tecla presionada.	`event.text()`, `event.key()`.
`pyqtSignal(tipos)`	Define una señal propia.	Atributo de clase de un `QObject`; nombre ASCII.
`señal.emit(args)`	Emite la señal.	Los `args` llegan al *slot*.

! Errores frecuentes

- Crear más de un `QApplication`, o crear un *widget* **antes** que la aplicación.
- Olvidar `show()` (no se ve nada) o `exec()` (el programa termina al instante).
- Crear *widgets* **sin padre** y sin meterlos en un *layout*: no se visualizan.
- Confundir las coordenadas: Y crece **hacia abajo**, y la posición de un hijo es relativa a su padre.
- Escribir `connect(self.metodo())` con paréntesis en lugar de `connect(self.metodo)`.
- Definir la `pyqtSignal` dentro de `__init__` (debe ser atributo de clase), en una clase que no hereda de `QObject`, o con tildes/`ñ` en el nombre.
- Llamar a `self.sender()` fuera de un *slot*.
- Mezclar lógica y dibujo en la misma clase, o llamar al *back-end* directamente desde el *front-end* (alto acoplamiento).
- Crear una ventana en una **variable local** y no guardarla: se descarta al terminar el método.

✓ Checklist de una GUI sana

1. ¿Hay **una sola** `QApplication`, creada antes de las ventanas, con `show()` y `exec()`? ¿Instalé el `sys.excepthook` para ver los errores?
2. ¿Cada *widget* tiene **padre** o está dentro de un *layout*? ¿Usé *layouts* en vez de posiciones fijas?
3. ¿Conecté las señales **sin paréntesis**? ¿Mis *slots* reciben los argumentos que la señal envía?
4. ¿Guardé una **referencia** (atributo o lista) de cada ventana que abro?
5. ¿Separé *front-end* y *back-end*? ¿Se comunican por **señales** y las conexiones están en un solo lugar?

Fin del resumen • ¡Que tus ventanas siempre respondan a tiempo!