

Unidad 2

Creación y uso de componentes.

DESARROLLO DE INTERFACES

2º DAM – IES POETA PACO MOLLÁ
CAMILO JUAN CASILLAS

Objetivos de aprendizaje

- ▶ ***Crear interfaces*** gráficas con ***Python***.
- ▶ ***Ubicar*** los ***componentes*** y ***modificar*** sus ***propiedades***.
- ▶ ***Asociar acciones*** a los acontecimientos de los componentes de la interfaz.
- ▶ ***Desarrollar aplicaciones*** con interfaz gráfica.

Índice

1. Introducción a Qt, PySide, PyQt
2. Lenguaje de programación Python
3. QT, PySide y PyQt
4. Interfaces de usuario a través de VSC
5. Ventana principal de QT, widgets y componentes comunes
6. Formas de organización. Uso de Layouts
7. Cuadros de diálogo, predeterminados o personalizados
8. Ventana principal, menús, barra de herramientas y docks flotantes
9. Creación y control de subventanas
10. Tematización. Estilos, paletas, iconos, ...
11. Diseñar con Qt Designer. Crear diseños con ayuda una interfaz grafica

“

1.- Introducción a Qt, PySide, PyQt.

”

FRAMEWORKS Y PUENTES PARA EMPEZAR A CREAR INTERFACES.

Entendamos qué vamos a utilizar para crear interfaces

1.- Introducción

- ▶ En esta segunda unidad vayamos a trabajar diferentes maneras de **crear interfaces gráficas**, haciendo uso de diferentes lenguajes de programación, diferentes frameworks e IDEs.
- ▶ Empezaremos con una breve introducción a la **creación de interfaces gráficas utilizando el lenguaje de programación Python** y la **biblioteca** específica para Python llamada **PySide** que permite **crear interfaces gráficas de Qt**.

“

2.- Lenguaje de programación Python.

”

FRAMEWORKS Y PUENTES PARA EMPEZAR A CREAR INTERFACES.

Entendamos qué vamos a utilizar para crear interfaces

2.- Python

- ▶ **Python** es un **lenguaje de programación de alto nivel, interpretado, orientado a objetos** y de **uso generalizado** con **semántica dinámica**, que se utiliza para la programación **de propósito general**.
- ▶ El nombre del lenguaje de programación Python proviene de una vieja serie de comedia de la BBC llamada Monty Python's Flying Circus.
- ▶ Dado que Monty Python es considerado uno de los dos nutrientes fundamentales para un programador (el otro es la pizza), el creador de Python nombró el lenguaje en honor al programa de televisión.

2.- Python

- ▶ Python fue **creado por Guido van Rossum**, nacido en 1956 en Haarlem, Países Bajos. Guido van Rossum **no desarrolló y evolucionó todos los componentes de Python**.
- ▶ La velocidad con la que Python **se ha extendido por todo el mundo** es el **resultado del trabajo continuo de miles de programadores, testers, usuarios y entusiastas**.
- ▶ Guido van Rossum **definió sus objetivos** para Python:
 - Un **lenguaje fácil, intuitivo, y muy poderoso**.
 - De **código abierto**, para que cualquiera pueda contribuir a su desarrollo.
 - El código que **es tan comprensible como el inglés simple**.
 - **Ideal para tareas cotidianas**, permitiendo tiempos de desarrollo cortos.

2.- Python

- ▶ **Es interpretado**, es decir, **se ejecuta en cualquier máquina que tenga un intérprete de Python**. Esto es una gran ventaja en la hora de hacer pequeños cambios, puesto que elimina la necesidad de recompilar código.
- ▶ Otras ventajas son: es un **lenguaje limpio y legible, combina el tipado fuerte con un tipado dinámico**.
- ▶ Python **se utiliza** actualmente entre otras cosas para **Big data, Data Science y Machine Learning**.

2.- Python

- ▶ Existen muchas **razones para utilizar** Python, algunas de ellas.
 - **Es fácil de aprender:** el tiempo necesario para aprender Python es más corto que en lenguajes.
 - **Es fácil de enseñar:** la carga de trabajo de enseñanza es menor que en otros lenguajes.
 - **Es fácil de utilizar para escribir software nuevo:** es posible escribir código más rápido cuando se emplea Python.
 - **Es fácil de entender:** es más fácil entender el código de otra persona más rápido si está escrito en Python.
 - **Es fácil de obtener, instalar y desplegar:** es **gratuito, abierto** y **multiplataforma**; no todos los lenguajes pueden presumir de eso.

2.- Python

- ▶ Existen **dos tipos principales de Python, llamados Python 2 y Python 3.**
- ▶ Python 2 es una versión anterior del Python original. Su **desarrollo se ha estancado intencionalmente**, aunque eso **no significa que no haya actualizaciones**. Por el contrario, las actualizaciones se emiten de forma regular, pero **no pretenden modificar el idioma de manera significativa**. Prefieren arreglar cualquier error recién descubierto y agujeros de seguridad. **La ruta de desarrollo de Python 2 ya ha llegado a un callejón sin salida, pero Python 2 en sí todavía está muy vivo.**
- ▶ **Python 3 es la versión más nueva**, la **actual**, del lenguaje. Está atravesando su **propio camino de evolución**, creando sus **propios estándares y hábitos**.
- ▶ Estas **dos versiones** de Python **no son compatibles entre sí**. Python 3 no es solo una versión mejorada de Python 2, **es un lenguaje completamente diferente**. Es **demasiado difícil, caro y arriesgado migrar una aplicación Python 2 antigua a una nueva plataforma**.

2.- Python

- **Python** es el **primer lenguaje del ranking del índice TIOBE** (octubre 2025) y tiene una amplia comunidad.

Oct 2025	Oct 2024	Change	Programming Language	Ratings	Change
1	1		 Python	24.45%	+2.55%
2	4	▲	 C	9.29%	+0.91%
3	2	▼	 C++	8.84%	-2.77%
4	3	▼	 Java	8.35%	-2.15%
5	5		 C#	6.94%	+1.32%
6	6		 JavaScript	3.41%	-0.13%
7	7		 Visual Basic	3.22%	+0.87%

“

3.- QT, PySide y PyQt.

”

FRAMEWORKS Y PUENTES PARA EMPEZAR A CREAR INTERFACES.

Entendamos qué vamos a utilizar para crear interfaces

3.- Qt, PySide, PyQt

- ▶ **Qt es un framework que se utiliza para programar interfaces gráficas de usuario** y está **programado con C++**. Es framework **multiplataforma, orientado a objetos, fácil de aprender**, software **libre, código abierto** y ofrece diferentes tipos de licencias.
- ▶ **PySide y PyQt son bindings o puentes que permiten utilizar Qt con Python.**
- ▶ **PySide es el binding oficial** (desarrollado por **The Qt Company**). Tiene **licencia pública gratuita**.
- ▶ **PyQt NO es lo binding oficial** (desarrollado por **Riverbank Computing**), tiene **licencia comercial de pago** para proyectos comerciales.

3.- Qt, PySide, PyQt

- ▶ **Qt es un framework de desarrollo de aplicaciones multiplataforma para escritorio y sistemas móviles.**
- ▶ **Sus desarrollos** permiten ejecutarse en plataformas como **Linux, OS X, Windows, Android, iOS...**
- ▶ **Qt NO es un lenguaje de programación, sino un conjunto de herramientas para el desarrollo de interfaces gráficas** de usuario multiplataforma hecho por C++.
- ▶ Algunos **ejemplos de aplicaciones desarrolladas con Qt** son:
 - Adobe **Photoshop Album**, para organizar imágenes.
 - El **escritorio KDE** de las distribuciones Linux.

3.- Qt, PySide, PyQt

- ▶ Algunos ejemplos de aplicaciones desarrolladas con Qt son:
 - **Last.fm Player**, el cliente de escritorio para streaming de música y radio.
 - **Skype** para mensajería y VOIP.
 - **TeamSpeak** para la comunicación con voz (gamers).
 - **VirtualBox** (virtualización de sistemas).
 - **LibreOffice**, suite ofimática libre (alternativa en Microsoft Office).
 - Otros: **Avidemux**, **Doxxygen**, **Scribus**, ...

3.- Qt, PySide, PyQt

- ▶ **Qt** es un framework **ampliamente utilizado** para desarrollar programas con interfaces gráficas de usuario:
 - Está **programado en C++**, por tanto, es muy rápido.
 - Es **multiplataforma**, funciona en diferentes sistemas operativos.
 - Es **orientado a objetos, fácil de empezar a utilizar y aprender**.
 - Es software **libre y código abierto**, su uso es seguro.
 - Ofrece **licencias públicas**, permitiendo su uso de forma **gratuita**.

“

4.- Interfaces de usuario a través de Visual Studio Code.

”

DEFINICIONES PRINCIPALES, CREAR DISEÑOS UTILIZANDO LENGUAJE PYTHON.

Crear interfaz a través de código

4.1.- Interfaz de usuario y Clases

- ▶ Como hemos visto anteriormente, **podemos** utilizar herramientas para la **creación de nuestra interfaz** de usuario, pero vamos a **comenzar utilizando código escrito en Python** para nuestros primeros pasos.
- ▶ Vamos a **ver** que **elementos** podemos utilizar, su **implementación**, así como, su **configuración** dependiendo de las diferentes opciones que nos permiten realizar cambios en los componentes.

4.1- Interfaz de usuario y Clases

- ▶ Repasemos un poco los **conceptos básicos de clases en Python** comparándolas con las clases en Java:
 - Código Java: **en java creamos una clase, con atributos, constructor y con los métodos** que creamos oportunos.
 - En Python:
 - **No se declaran tipos de datos** (Python es dinámico).
 - **__init__ es el constructor** (se ejecuta al crear el objeto, y **su primer parámetro es self, que es el mismo**)
 - **self equivale a this a Java.**
 - **No utilizamos new para crear una instancia** del objeto.
 - **No es necesario un main()**: el código se puede ejecutar directamente.

4.1- Interfaz de usuario y Clases

➤ Código Java

VS

Código Python:

```
1 public class Persona {  
2     // Atributos  
3     private String nombre;  
4     private int edad;  
5     private String ciudad;  
6  
7     // Constructor  
8     public Persona(String nombre, int edad, String ciudad) {  
9         this.nombre = nombre;  
10        this.edad = edad;  
11        this.ciudad = ciudad;  
12    }  
13  
14     // Método  
15     public void saludar() {  
16         System.out.println("Hola, soy " + nombre +  
17             ", tengo " + edad + " años y vivo en "  
18             + ciudad + ".");  
19     }  
20 }  
21  
22 // Método principal para probar  
23 public static void main(String[] args) {  
24     Persona personal = new Persona("Iván", 40, "Alcoi");  
25     personal.saludar();  
26 }
```

```
1 class Persona:  
2     def __init__(self, nombre, edad, ciudad):  
3         self.nombre = nombre  
4         self.edad = edad  
5         self.ciudad = ciudad  
6  
7     def saludar(self):  
8         print(f"Hola, soy {self.nombre},  
9             tengo {self.edad} años y vivo en  
10            {self.ciudad}.")  
11  
12     # Crear objeto  
13 personal = Persona("Iván", 40, "Alcoi")  
14 personal.saludar()
```

4.1- Interfaz de usuario y Clases

- ▶ El código para **crear los interfaces** a utilizar, están **basados en la utilización de clases y objetos**, para hacerlo **necesitaremos** trabajar con la **herencia** de clases.
- ▶ Repasemos un poco los **conceptos básicos**:

- En Python, **cuando una clase** (llamada subclase) **hereda de otra** (llamada superclase) **obtiene todo su comportamiento**.

```
class Madre:  
    def __init__(self):  
        print(f"Soy Madre")  
  
class Hijo(Madre):  
    pass  
  
hijo = Hijo()
```

```
[Running] python3  
Soy Madre
```

Código [1-herencia.py](#)

4.1- Interfaz de usuario y Clases

- La **subclase** puede **sobrescribir los métodos de la superclase** para realizar sus propias acciones:

```
class Madre:
    def __init__(self):
        print(f"Soy Madre")

class Hijo(Madre):
    def __init__(self):
        super().__init__()
        print(f"Soy Hijo")

hijo = Hijo()

[Running] python3
Soy Madre
Soy Hijo
```

```
class Madre:
    def __init__(self):
        print(f"Soy Madre")

class Hijo(Madre):
    def __init__(self):
        print(f"Soy Hijo")
```

```
[Running] python3
Soy Hijo
```

- Hay **ocasiones** en las que **nos interesa no sobrescribir completamente, sino extender una funcionalidad de la superclase**. Cuando necesitemos este comportamiento podemos hacer uso de la función `super()`, un acceso directo a la superclase.
- Es **possible extender el comportamiento de una superclase sin sobrescribirlo si hacemos uso del acceso `super()`**

Código [2-sobrescribir-py](#)
[3-extender.px](#)

4.1- Interfaz de usuario y Clases

- Debemos tener **cuidado con la herencia múltiple**, ya que al utilizar el acceso super() con más de una clase donde heredar, **se sigue la lógica de la prioridad de herencia, teniendo más prioridad la clase de la izquierda**, y por lo tanto super() es el acceso de Madre :

```
class Hijo(Madre, Padre):
    def __init__(self):
        Padre.__init__(self)
        print(f"Soy Hijo")
```

[Running] python3
Soy Padre
Soy Hijo

- **Si quisiéramos extender el comportamiento del padre, en lugar de super() utilizaremos su propio nombre:**
- Sin embargo, **aquí deberemos pasar self al método porque super lo hace implícitamente.**

```
class Madre:
    def __init__(self):
        print(f"Soy Madre")

class Padre:
    def __init__(self):
        print(f"Soy Padre")

class Hijo(Madre, Padre):
    def __init__(self):
        super().__init__()
        print(f"Soy Hijo")
```

[Running] python3
Soy Madre
Soy Hijo

4.1- Interfaz de usuario y Clases

- Por último, nada nos impediría **extender el funcionamiento tanto de la madre como del padre**:

```
class Hijo(Madre, Padre):  
    def __init__(self):  
        Madre.__init__(self)  
        Padre.__init__(self)  
        print(f"Soy Hijo")
```

```
[Running] python3  
Soy Madre  
Soy Padre  
Soy Hijo
```

- Con esto queda repasado el concepto de clases, objetos y herencia múltiple, necesario para extender los componentes de PySide.

4.2- Aplicación base (QApplication)

- ▶ En el contexto de desarrollo con Qt, tenemos la **función principal con QApplication**, que es el **núcleo de cualquier programa de Qt y es responsable de gestionar el ciclo de vida de la aplicación**.
- ▶ Existe un **bucle de eventos**, gestionado por QApplication, **encargado de gestionar todas las interacciones con la interfaz gráfica de usuario**.
- ▶ Cuando **se cierre de la aplicación, QApplication termina**, todas las ventanas y componentes de la aplicación se cierran.

4.2- Aplicación base (QApplication)

- ▶ Una **aplicación requiere** por lo menos **un widget para mostrar algo en pantalla**, así que la estructura más básica de un programa es utilizar un widget vacío.

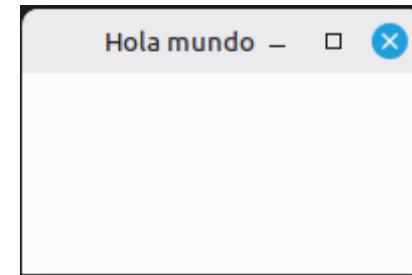
```
1 import sys
2 from PySide6.QtWidgets import QApplication, QWidget
3
4
5 # Creamos una aplicación para gestionar la interfaz
6 app = QApplication(sys.argv)
7
8 # Creamos un widget para generar la ventana
9 window = QWidget()
10
11 # Mostramos la ventana, se encuentra oculta por defecto
12 window.show()
13
14 # Iniciamos el bucle del programa
15 sys.exit(app.exec_())
```



4.2- Aplicación base (QApplication)

- ▶ Un solo Widget no es muy útil, ya que toda la ventana es un panel en sí mismo, por suerte, **Qt nos ofrece un widget capaz de gestionar ventanas con multitud de funcionalidades**, se llama **QMainWindow**, el **widget para gestionar ventanas principales**.
- ▶ Todos **los widgets que heredan de QWidget** se pueden visualizar como **ventanas en sí mismos**.

```
1 import sys
2 from PySide6.QtWidgets import QApplication, QMainWindow
3
4 app = QApplication(sys.argv)
5
6 # Ahora la ventana la gestiona el widget de ventana principal
7 window = QMainWindow()
8
9 # Damos un título a la ventana principal
10 window.setWindowTitle("Hola mundo")
11
12 window.show()
13
14 sys.exit(app.exec_())
```



Código [8-holamundo.py](#)

4.2- Aplicación base (QApplication)

- ▶ **Aparentemente tenemos lo mismo que al usar un QWidget, pero esta ventana principal nos permite asignar un widget para ocupar su espacio central:**

```
1  from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton  
2  import sys  
3  
4  app = QApplication(sys.argv)  
5  window = QMainWindow()  
6  window.setWindowTitle("Hola mundo")  
7  
8  # Guardamos el botón en una variable  
9  button = QPushButton("Soy un botón")  
10 # Establecemos el botón como widget central de la ventana principal  
11 window.setCentralWidget(button)  
12  
13 window.show()  
14 sys.exit(app.exec_())
```



Código [9-botón.py](#)

4.2- Aplicación base (QApplication)

- Veamos como cambia nuestro código Python ***si utilizamos clases y objetos:***

```

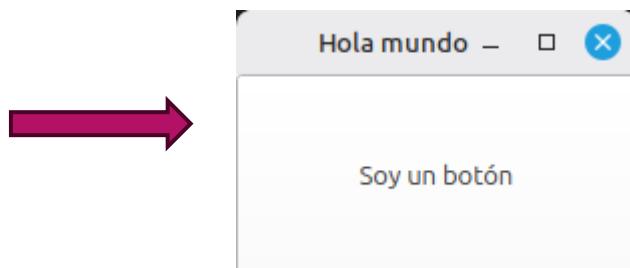
1  from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
2  import sys
3
4
5  class MainWindow(QMainWindow):
6
7      """
8          Creamos nuestra propia clase MainWindow heredando de QMainWindow
9      """
10
11     # Creamos la ventana en el constructor a partir de una QMainWindow
12     def __init__(self):
13
14         # Con super ejecutamos su propio constructor
15         # Así se crea la ventana en su propia instancia self
16         super().__init__()
17
18         # Damos un título al programa
19         self.setWindowTitle("Hola mundo")
20
21         # Guardamos el botón en una variable
22         button = QPushButton("Hola")
23
24         # Establecemos el botón como widget central de la ventana principal
25         self.setCentralWidget(button)
26

```

```

27
28     # Si ejecutamos el propio script como programa principal
29     if __name__ == "__main__":
30         # Creamos la aplicación
31         app = QApplication(sys.argv)
32         # Creamos nuestra ventana principal
33         window = MainWindow()
34         # Mostramos la ventana
35         window.show()
36         # Iniciamos el bucle del programa
37         sys.exit(app.exec_())

```



Código [10-mainpoo.py](#)

4.2- Aplicación base (QApplication)

- ▶ **La clave** es **extender el funcionamiento del constructor** de **QMainWindow**, pues al ejecutar super().__init__() heredamos su comportamiento.
- ▶ Desde ese momento, la propia instancia self de nuestra clase **MainWindow adquiere los métodos heredados como setWindowTitle y setCentralWidget**.

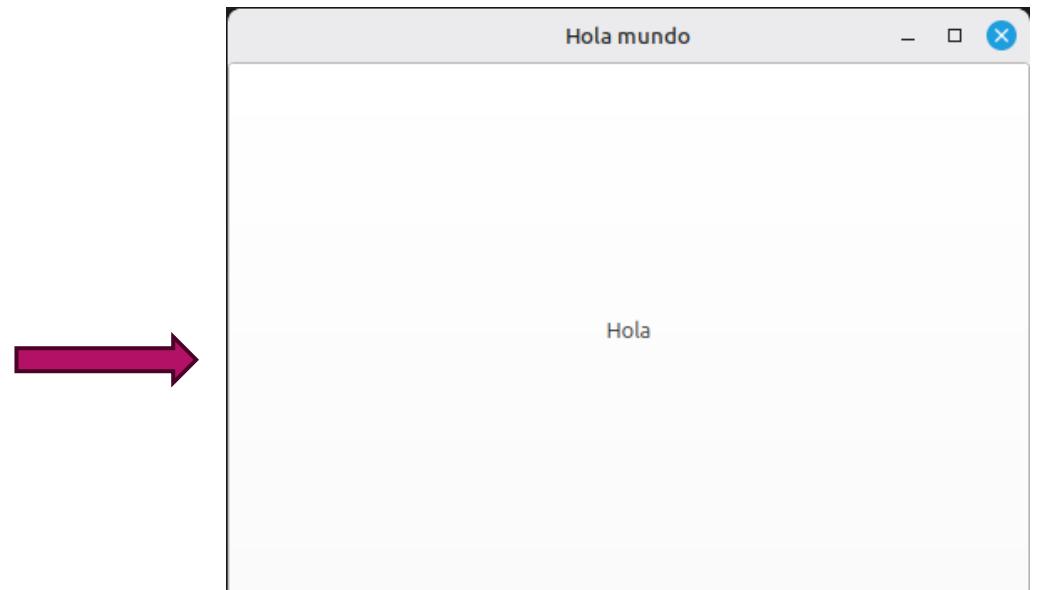
4.3- Clases vs métodos

- ▶ Debemos tener claro que, ***al tratar con clases, tenemos métodos propios o heredados*** que podemos utilizar para configurar nuestros objetos a la forma deseada.
- ▶ En muchos casos, ***existe también algunos “objetos” de la librería, que podemos utilizar*** para configurar objetos de nuestras clases.
- ▶ En la mayoría de los casos podemos encontrar las dos versiones/opciones, la clase suele poseer muchas más características de configuración que los métodos propios/heredados, pero ***es elección del usuario y de sus necesidades el uso de cualquiera de las opciones*** de las que disponga.
- ▶ Vamos a ver un ejemplo, en el caso del tamaño de nuestras ventanas que nos hará comprender mejor el uso de las dos opciones.

4.3- Clases vs métodos

- ▶ Podemos utilizar el **método resize de la ventana, para configurar sus dimensiones**. El inconveniente es que este método no establece las reglas independientes del objeto.

```
1 from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton  
2 import sys  
3  
4 class MainWindow(QMainWindow):  
5     def __init__(self):  
6         super().__init__()  
7         self.setWindowTitle("Hola mundo")  
8         button = QPushButton("Hola")  
9         self.setCentralWidget(button)  
10  
11         # Redimensión simple  
12         self.resize(480, 320)  
13  
14  
15 if __name__ == "__main__":  
16     app = QApplication(sys.argv)  
17     window = MainWindow()  
18     window.show()  
19     sys.exit(app.exec_())
```

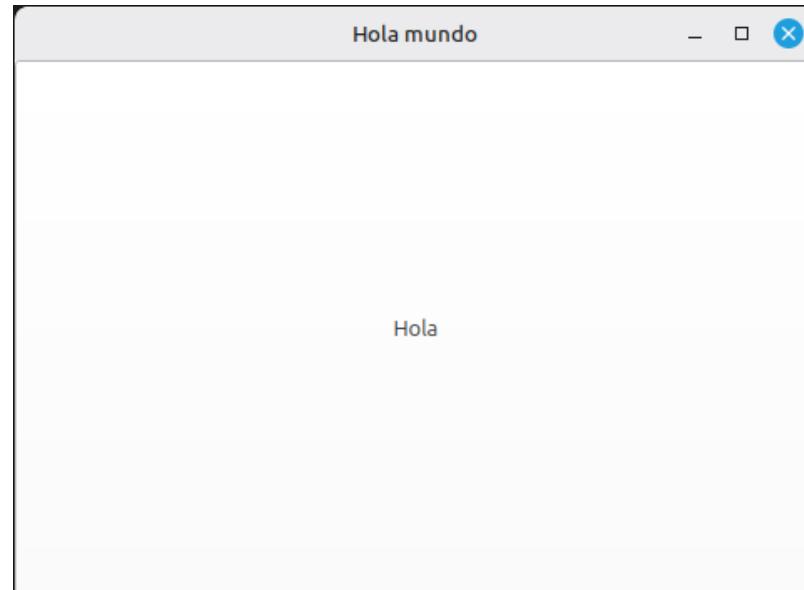


Código [11-tamano.py](#)

4.3- Clases vs métodos

- ▶ La otra opción es que **en Qt existe un objeto llamado QSize** que podemos **asignar a los widgets para controlar su tamaño**. Toma un ancho y un alto en píxeles y se puede establecer como tamaño mínimo, máximo o fijo para un widget:

```
1 from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
2 from PySide6.QtCore import QSize # Nuevo
3 import sys
4
5 class MainWindow(QMainWindow):
6     def __init__(self):
7         super().__init__()
8         self.setWindowTitle("Hola mundo")
9         button = QPushButton("Hola")
10        self.setCentralWidget(button)
11
12        # Tamaño mínimo del widget
13        self.setMinimumSize(QSize(480, 320))
14        # Tamaño máximo del widget
15        self.setMaximumSize(QSize(480, 320))
16        # Tamaño fijo del widget
17        self.setFixedSize(QSize(480, 320))
18
19
20 if __name__ == "__main__":
21     app = QApplication(sys.argv)
22     window = MainWindow()
23     window.show()
24     sys.exit(app.exec_())
```



Código [12-tamanyomaxmin.py](#)

4.4- Señales y receptores

- ▶ En **Qt para añadir una funcionalidad a un Widget necesitamos conectarlo a una acción**, algo que se consigue mediante **señales y slots**.
- ▶ **Las señales son notificaciones emitidas por los widgets cuando sucede algo.** Por ejemplo, un botón envía una señal de "botón pulsado" cuando un usuario hace clic en él.
- ▶ Aunque de poco sirve que un widget envíe una señal si nadie es consciente de ello. Para ese propósito **existen los receptores, conocidos como slots**.

4.4- Señales y receptores

- ▶ En Qt, **las señales, se combinan con los slots**, que son las **funciones que reciben la notificación y ejecutan una acción**. El sistema de señales y slots de Qt permite conectar estas señales a slots para que la aplicación responda a eventos de manera segura y eficiente.
- ▶ Características principales:
 - **Notificación de eventos:** las señales son emitidas por un objeto (el "emisor") para notificar sobre un evento específico.
 - **Recepción de notificaciones:** los slots son funciones (en el objeto "receptor") que se conectan a una señal y se ejecutan cuando esta es emitida.
 - **Conexión flexible:** se pueden conectar una señal a uno o más slots, y un slot puede ser conectado a una o más señales.

4.4- Señales y receptores

- **Seguridad de tipos:** El sistema de **señales y slots es seguro** en cuanto a tipos, lo que significa que las señales y slots que se conectan **deben tener tipos de argumentos compatibles**.
- **Comunicación entre hilos:** Las señales son seguras para subprocessos y pueden usarse para la comunicación segura entre diferentes hilos.
- ▶ Ejemplos de **uso**.
 - **Botones:** al hacer **clic en un botón**, este emite la señal `clicked()`. Se conecta esta señal a un slot, **ejecutar una función** y responder al clic.
 - **QAction:** se utiliza para **acciones de menú o barras de herramientas**. Emite señales como `triggered()` cuando la acción es activada.
 - **QThread:** **emiten señales `started()` cuando un hilo está listo para ejecutar código y `finished()` cuando ha terminado.**

4.4- Señales y receptores

- ▶ En el contexto de Qt, "**receptores**" slots, son métodos que reciben y procesan las señales emitidas por otros objetos. Cuando un evento ocurre emite una señal, y los slots conectados actúan como receptores, ejecutando el código que les corresponde. Es el mecanismo principal para la comunicación entre objetos, permitiendo la creación de aplicaciones dinámicas y con interfaz gráfica de manera organizada
- ▶ Respecto a los acontecimientos:
 - Cada interacción del usuario con la interfaz, por ejemplo, un clic de ratón, una pulsación de tecla... generará un acontecimiento. Este acontecimiento será añadido a una cola de acontecimientos (event queue) para ser gestionado.

4.4- Señales y receptores

- ▶ Respecto a los acontecimientos:
 - ***El bucle de acontecimientos*** (event loop), que ***es un bucle infinito, comprobará en cada iteración si hay acontecimientos pendientes de ser gestionados.*** En caso de que sí, el acontecimiento será gestionado por el gestor de acontecimientos (event handler) que ejecutará su “manejador”.
 - ***El bucle de acontecimientos será gestionado por el objeto QApplication*** y lanzará al ejecutar el método exec() de este.

4.5- Manipular componentes

- ▶ Si deseamos acceder a un widget desde un método es tan sencillo como **almacenar un acceso a ese widget en la propia instancia**, es decir, **usar un atributo de clase**:

variable = widget_deseado

```
# Mi vble boton apunta a un widget QPushButton  
boton = QPushButton("Hola")
```

- ▶ Una vez contamos con **el puntero**, podemos hacer referencia a cualquier instancia de un widget para **modificarla a voluntad**

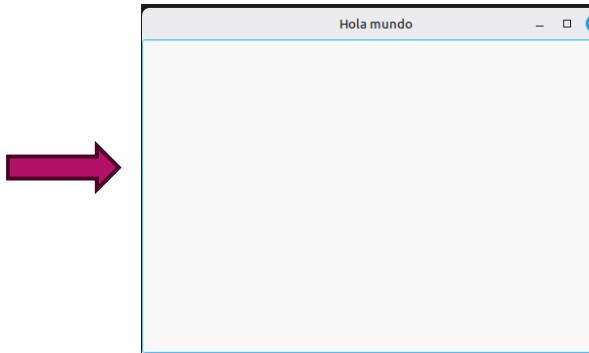
```
boton.setText("Adios")  
boton.icon() # Método para obtener el icono (vacío en este caso)
```

4.5- Manipular componentes

► Respecto a los acontecimientos:

```
1  from PySide6.QtWidgets import QApplication, QMainWindow, QLineEdit # editado
2  from PySide6.QtCore import QSize
3  import sys
4
5
6  class MainWindow(QMainWindow):
7      def __init__(self):
8          super().__init__()
9          self.setWindowTitle("Hola mundo")
10         self.setMinimumSize(QSize(480, 320))
11
12         # widget input de texto
13         texto = QLineEdit()
14         # capturamos la señal de texto cambiado
15         texto.textChanged.connect(self.texto_modificado)
16
17         # establecemos el widget central
18         self.setCentralWidget(texto)
19
20         # creamos el puntero
21         self.texto = texto
```

```
23     def texto_modificado(self):
24         # recuperamos el texto del input
25         texto_recuperado = self.texto.text()
26         # modificamos el título de la ventana al vuelo
27         self.setWindowTitle(texto_recuperado)
28
29
30     if __name__ == "__main__":
31         app = QApplication(sys.argv)
32         window = MainWindow()
33         window.show()
34         sys.exit(app.exec_())
```



Código [13-manipulable.py](#)

“

5.- Ventana principal de QT, widgets y componentes comunes.

”

CREAMOS UNA INTERFAZ DE USUARIO DESDE CERO.

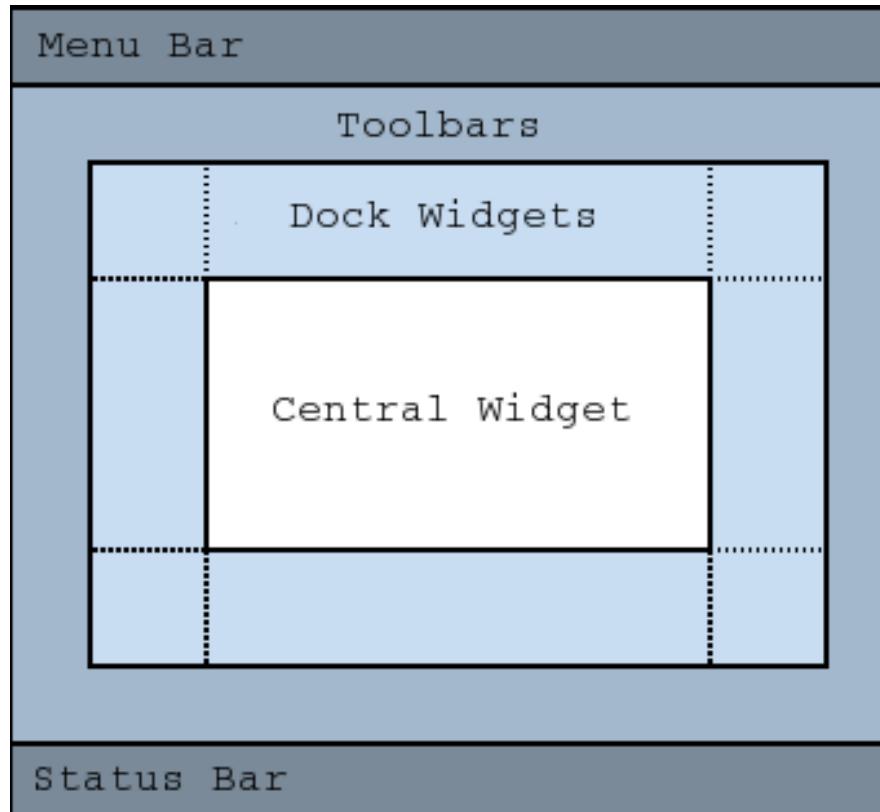
Creación y configuración de elementos

5.1- Marco de ventana principal de Qt

- ▶ Una **ventana principal** proporciona un marco **para construir la interfaz de usuario de una aplicación**.
- ▶ Qt tiene **QMainWindow** y sus **clases relacionadas** para la **gestión de ventanas principales**.
- ▶ **QMainWindow tiene su propio diseño** al que puede agregar **QToolBars, QDockWidgets, a QMenuBar, y a QStatusBar**.
- ▶ El diseño tiene un **área central** que puede ser **ocupada** por **cualquier tipo de widget**. Puede ver una imagen del diseño a continuación.

5.1- Marco de ventana principal de Qt

- ▶ Imagen del **diseño de una ventana principal**:



5.1 - Marco de ventana principal de Qt

- ▶ **Un widget central suele ser un widget estándar** de Qt, como un **QTextEdit** o un **QGraphicsView**. También se pueden usar **widgets personalizados** para aplicaciones avanzadas. El widget central **se configura con setCentralWidget()**.
- ▶ Las **ventanas principales tienen una interfaz de documento único (SDI) o múltiple (MDI)**. Las aplicaciones MDI en Qt se crean usando un QMdiArea como widget central.

5.2- El Widget

- ▶ ***El widget es el átomo de la interfaz de usuario: recibe el ratón, el teclado y otros eventos del sistema de ventanas, y pinta una representación de sí mismo en la pantalla.***
- ▶ Cada ***widget es rectangular y se clasifican en un orden Z***. Un widget ***es recortado por su padre y por los widgets frente a él.***
- ▶ ***Un widget que no está incrustado en un widget padre se llama ventana.*** Por lo general, las ventanas ***tienen un marco y una barra de título***, aunque también es posible crear ventanas sin dicha decoración utilizando banderas de ventana adecuadas.

5.2- El Widget

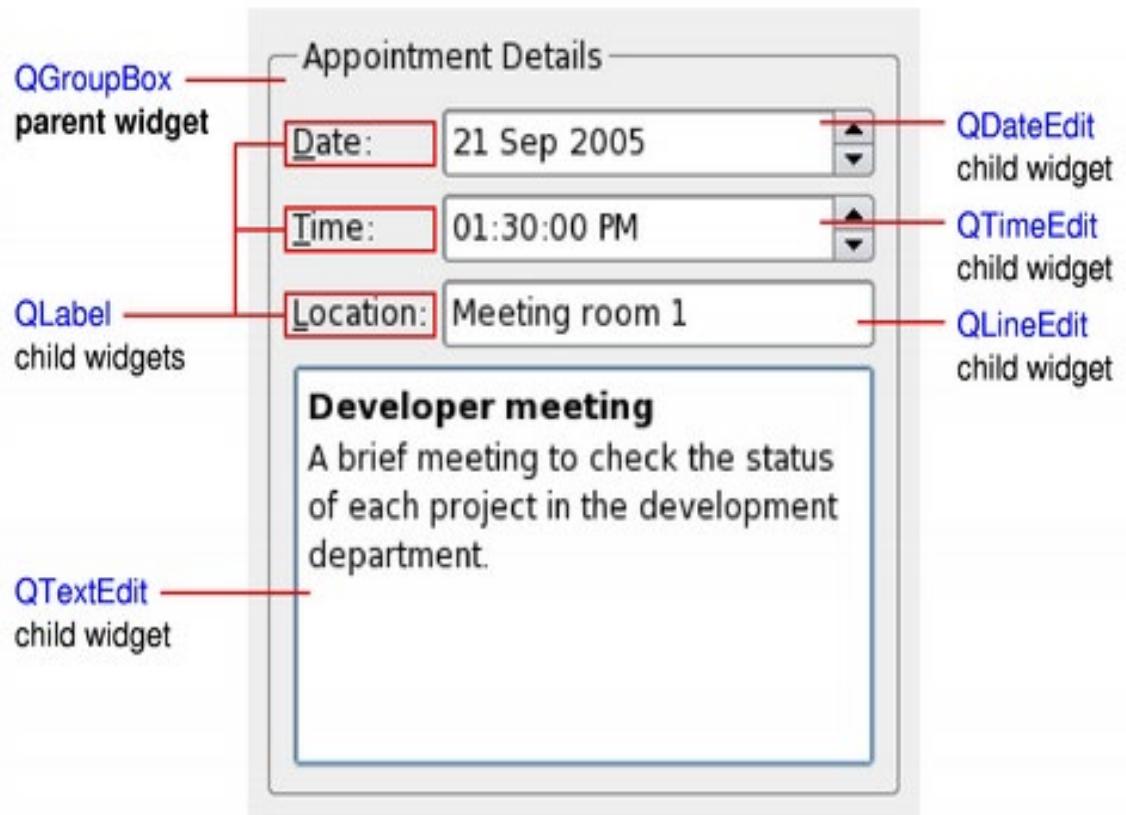
- ▶ En Qt, **QMainWindow** y las diversas **subclases de QDialogSon los tipos** de ventanas **más comunes**.
- ▶ **QWidget** tiene **muchas funciones miembro**, pero **algunas** de ellas tienen **poca funcionalidad** directa; por ejemplo, una propiedad de fuente, pero que es muy poco usada.
- ▶ Hay **muchas subclases** que **proporcionan una funcionalidad real, como QLabel, QPushButton, QListWidget, y QTabWidget**.
- ▶ **Un widget sin un widget padre**, es siempre **una ventana independiente** (widget de nivel superior). Para estos widgets, **setWindowTitle()** y **setWindowIcon()** establecer la barra de **título** y el **ícono**, respectivamente.

5.2- El Widget

- ▶ Los widgets que no son ventana principal, son **widgets secundarios**, que **se muestran dentro de sus widgets padre**.
- ▶ **La mayoría de los widgets en Qt son principalmente útiles como widgets hijos.** Por ejemplo, es posible mostrar un botón como una ventana de nivel superior, pero la mayoría de la gente prefiere poner sus botones dentro de otros widgets, como QDialog.

5.2- El Widget

- ▶ Este diagrama muestra una QGroupBox, widget utilizado para contener varios widgets hijos en un diseño proporcionado por QGridLayout.
- ▶ Si quieres utilizar un QWidget para mantener los widgets secundarios, por lo general querrás agregar un diseño al padre QWidget, que veremos en el apartado de Layouts.

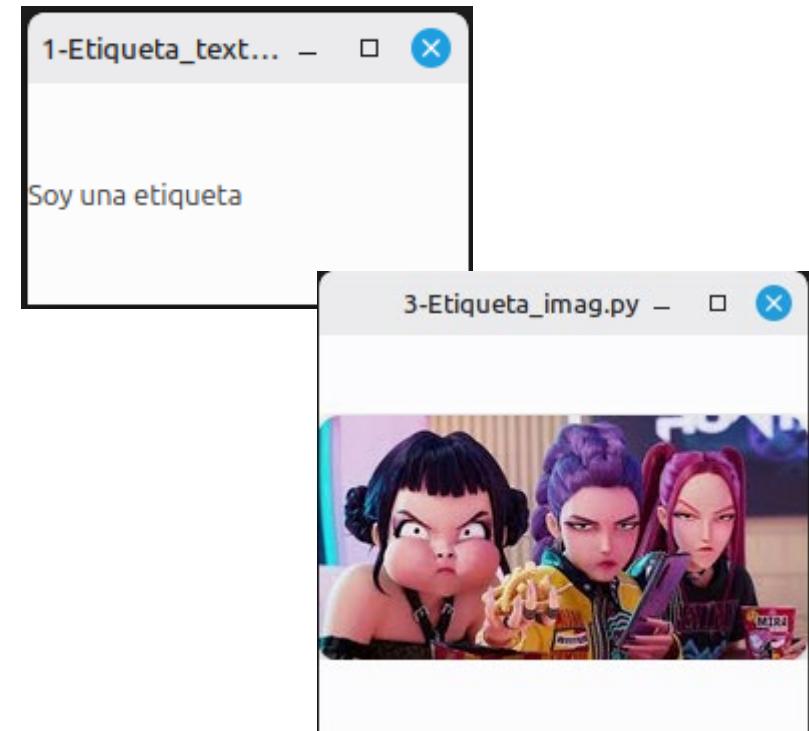


5.2- El Widget

- ▶ Cuando se utiliza **un widget como contenedor para agrupar un número de widgets secundarios**, se conoce como un **widget compuesto**. Estos **se pueden crear mediante la construcción de un widget** con las propiedades visuales requeridas - a QFrame, por ejemplo - **y añadir widgets hijos** a la misma, generalmente gestionados por un diseño.
- ▶ **Los widgets compuestos también se pueden crear subclasiificando un widget estándar, como QWidget o QFrame, y añadiendo** el diseño necesario y los **widgets secundarios en el constructor de la subclase**. Muchos de los ejemplos proporcionados con Qt utilizan este enfoque.

5.3- Etiqueta QLabel

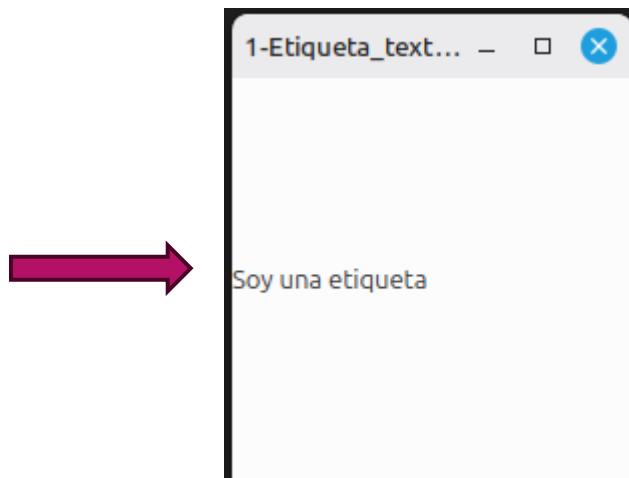
- ▶ Uno de los **widgets más sencillos de Qt**.
- ▶ Se trata de una **pieza de texto que se puede posicionar** en nuestra aplicación.
- ▶ Podemos **asignar el texto al crearlas o mediante el método setText()**
- ▶ Esta etiqueta **también admite una imagen**.



5.3- Etiqueta QLabel

- ▶ Un **ejemplo de creación** de este componente podría ser el siguiente código:

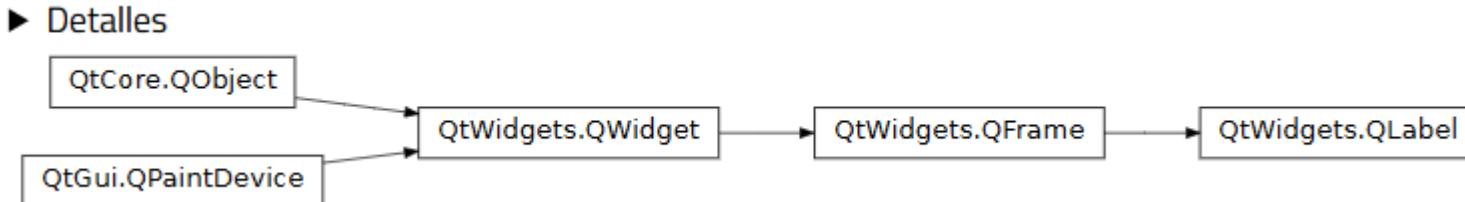
```
1  from PySide6.QtWidgets import QApplication, QMainWindow, QLabel
2  from PySide6.QtCore import QSize
3  import sys
4
5
6  class MainWindow(QMainWindow):
7      def __init__(self):
8          super().__init__()
9          self.setMinimumSize(QSize(200, 200))
10
11         # widget etiqueta
12         etiqueta = QLabel("Soy una etiqueta")
13
14         self.setCentralWidget(etiqueta)
15
16
17     if __name__ == "__main__":
18         app = QApplication(sys.argv)
19         window = MainWindow()
20         window.show()
21         sys.exit(app.exec_())
```



Código [1-Etiqueta_texto.py](#)

5.3- Etiqueta QLabel

- ▶ Como se trata de nuestro **primer elemento estudiado en profundidad** y es muy sencillo de entender, vamos a ver que podemos encontrar en la clase QLabel:
- ▶ Es una **clase que hereda de QFrame, que a su vez, hereda de QWidget, ...**



- ▶ Posee **estos atributos:**

- `alignment` - La alineación del contenido de la etiqueta
- `hasSelectedText` - Si hay algún texto seleccionado
- `indent` - Sangría del texto de la etiqueta en píxeles
- `margin` - El ancho del margen
- `openExternalLinks`
- `pixmap` - El mapa de píxeles de la etiqueta
- `scaledContents` - Si la etiqueta ajustará su contenido para ocupar todo el espacio disponible
- `selectedText` - El texto seleccionado
- `text` - El texto de la etiqueta
- `textFormat` - Formato del texto de la etiqueta
- `textInteractionFlags`
- `wordWrap` - La política de la discográfica sobre el ajuste de palabras

5.3- Etiqueta QLabel

► Los **métodos** disponibles **en la clase** son:

- `def __init__()`
- `def alignment()`
- `def buddy()`
- `def hasScaledContents()`
- `def hasSelectedText()`
- `def indent()`
- `def margin()`
- `def movie()`
- `def openExternalLinks()`
- `def picture()`
- `def pixmap()`
- `def selectedText()`
- `def selectionStart()`
- `def setAlignment()`
- `def setBuddy()`
- `def setIndent()`
- `def setMargin()`
- `def setOpenExternalLinks()`
- `def setScaledContents()`
- `def setSelection()`
- `def setTextFormat()`
- `def setTextInteractionFlags()`
- `def setWordWrap()`
- `def text()`
- `def textFormat()`
- `def textInteractionFlags()`
- `def wordWrap()`

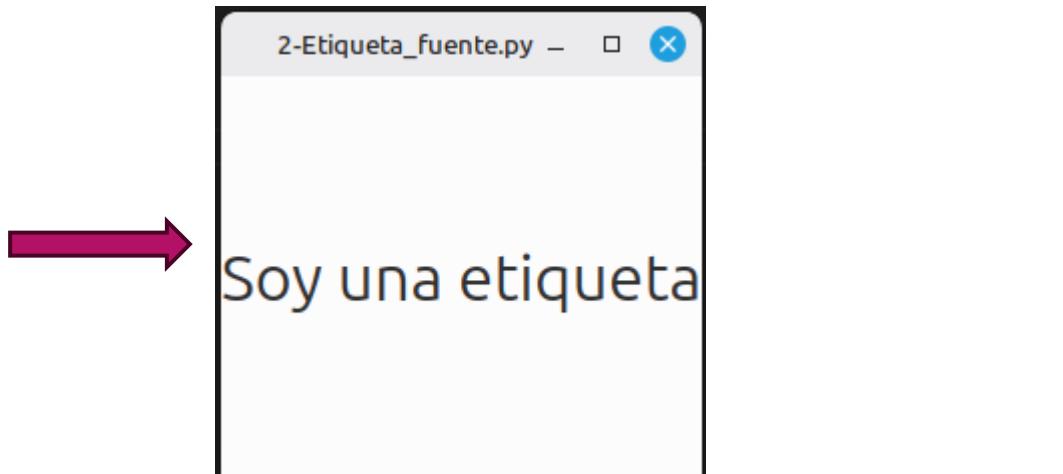
5.3- Etiqueta QLabel

- ▶ Vemos los *Slot*, que son **funciones que se ejecutan en respuesta a una señal emitida por otro objeto:**
 - def `clear()`
 - def `setMovie()`
 - def `setNum()`
 - def `setPicture()`
 - def `setPixmap()`
 - def `setText()`
- def `linkActivated()`
- def `linkHovered()`
- ▶ Por último, **señales que puede enviar este objeto y que deben provocar alguna respuesta:**

5.3- Etiqueta QLabel

- ▶ Vamos a ejecutar algunos **ejemplos** utilizando algunos **atributos** o **métodos** vistos.
- ▶ En primer lugar, podemos **configurar una fuente a través de la cuál controlar el tamaño y otros atributos**. Podemos **recuperar la fuente** por defecto y aumentar su tamaño. Fíjate en los atributos y métodos utilizados, de donde provienen:

```
# widget etiqueta
etiqueta = QLabel("Soy una etiqueta")
# recuperamos la fuente por defecto, atributo heredado de QWidget
# y devuelve un objeto QFont
fuente = etiqueta.font()
# establecemos un tamaño, con un metodo de QFont
fuente.setPointSize(24)
# la asignamos a la etiqueta, con un metodo de QWidget
etiqueta.setFont(fuente)
```

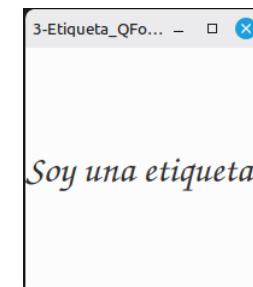


Código [2-Etiqueta_fuente.py](#)

5.3- Etiqueta QLabel

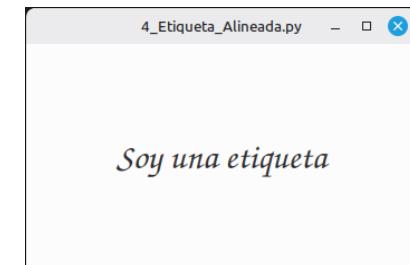
- Vamos a repetir el ***mismo ejemplo***, pero ***ahora vamos a crear una instancia de la clase QFont***

```
# Añadimos la libreria "from PySide6.QtGui import QFont # nuevo"
# cargamos una fuente del sistema, creando un objeto QFont
fuente = QFont("Comic Sans MS", 24)
# la asignamos a la etiqueta, con un metodo de QWidget
etiqueta.setFont(fuente)
```



- Las ***etiquetas*** también nos permiten ***alinearlas respecto a su contenedor***, para ello necesitamos importar las ***definiciones estándar de Qt***

```
# Añadimos la libreria "from PySide6.QtCore import QSize, Qt # editado"
# establecemos unas flags de alineamiento
etiqueta.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
```

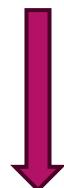


Código [3-Etiqueta_QFont.py](#)
[4_Etiqueta_Alineada.py](#)

5.3- Etiqueta QLabel

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

```
# mostramos los valores enteros y binarios de las constantes
print(int(Qt.AlignHCenter), int(Qt.AlignVCenter), int(Qt.AlignHCenter | Qt.AlignVCenter))
print(bin(Qt.AlignHCenter), bin(Qt.AlignVCenter), bin(Qt.AlignHCenter | Qt.AlignVCenter))
```



```
(ModuloDI) alu@Pc-IAW:~/DI_UD2/ModuloDI$ /home/alu/DI_UD2/ModuloDI/bin/python /home/alu/DI_UD2/ModuloDI/Apartado5/5-Etiqueta_Flags.py
4 128 132
0b100 0b10000000 0b10000100
```



Código [5-Etiqueta_Flags.py](#)

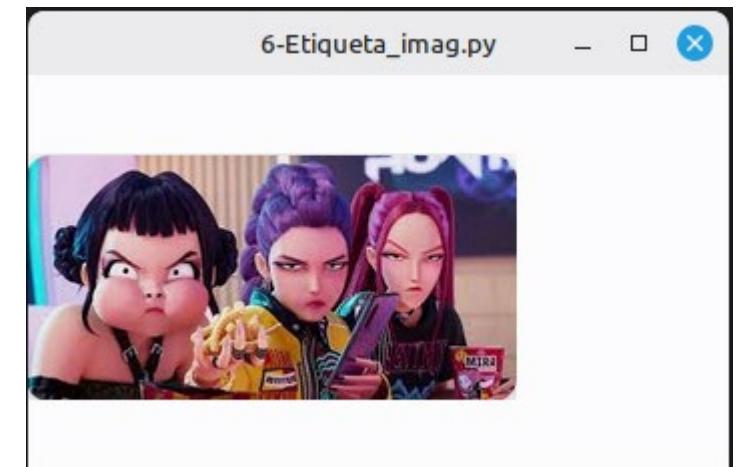
5.3- Etiqueta QLabel

- ▶ Si nos fijamos bien en lo que ha ocurrido en la ejecución del código anterior, podemos ver que ***la tubería es la unión de conjuntos, es decir que une dos opciones o más.***
- ▶ Podemos ver como colocar el texto en una etiqueta de ***forma sencilla utilizando estos flags.***
 - `Qt.AlignHCenter | Qt.AlignVCenter` → texto centrado totalmente.
 - `Qt.AlignHCenter | Qt.AlignTop` → texto centrado horizontalmente, pero arriba.
 - `Qt.AlignLeft | Qt.AlignTop` → texto arriba a la izquierda.
 - `Qt.AlignRight | Qt.AlignBottom` → texto abajo a la derecha.

5.3- Etiqueta QLabel

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

```
# incluimos la libreria "from PySide6.QtGui import QPixmap # editado"
# creamos la imagen, creando una instancia de QPixmap
imagen = QPixmap("./Imagenes/k-pop.png")
# la asignamos a la etiqueta
etiqueta.setPixmap(imagen)
# establecemos el widget central
self.setCentralWidget(etiqueta)
```



Código [6-Etiqueta_imag.py](#)

5.3- Etiqueta QLabel

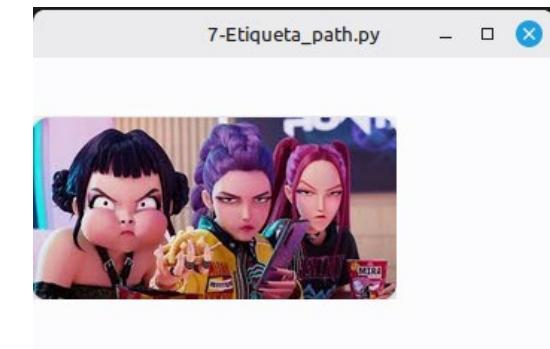
- ▶ En el momento en que cargamos recursos externos **debemos** empezar a **tener en cuenta el concepto de la ruta al recurso**.
- ▶ Cuando establecemos la imagen.png, **el programa espera que ese recurso se encuentre en el mismo directorio desde donde se ejecuta el script**. Aunque como vemos cuando ejecutamos en Visual Studio Code, nuestro directorio raíz es donde esta activada nuestro entorno virtual MODULODI. Por eso debemos tener en cuenta que es posible ejecutar un script sin estar en su mismo directorio y si lo hacemos esos recursos no se encontrarán.
- ▶ Alternativamente **podemos utilizar el módulo Path de Python para generar una ruta absoluta al recurso concreto** a partir del script actual y solventar el problema para siempre.

5.3- Etiqueta QLabel

- ▶ Os recomiendo crear una función como la siguiente para facilitar la reutilización:

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

# incluimos la libreria "from PySide6.QtGui import QPixmap # editado"
# incluimos la libreria "from pathlib import Path"
# creamos la imagen, creando una instancia de QPixmap
print(absPath("./Imagenes/k-pop.png"))
imagen = QPixmap(absPath("./Imagenes/k-pop.png"))
# la asignamos a la etiqueta
etiqueta.setPixmap(imagen)
# establecemos el widget central
self.setCentralWidget(etiqueta)
```



- ▶ Para finalizar, si quisiéramos que se reescalase junto al tamaño de la ventana deberíamos establecer el atributo scaledContents en True:

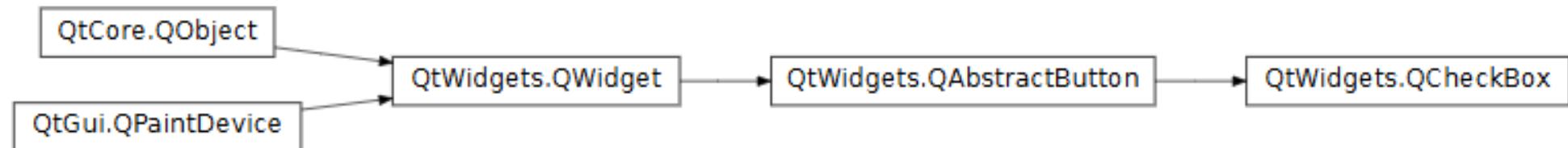
```
# hacemos que se escale con la ventana
etiqueta.setScaledContents(True)
```



Código [7-Etiqueta_path.py](#)
[8-Etiqueta_Escalable.py](#)

5.4- QCheckbox (Casillas de verificación)

- ▶ Un **QCheckBox** es un **widget** de interfaz gráfica que **permite** al usuario **seleccionar o deseleccionar una opción**.
- ▶ Se utiliza para que **el usuario elija una o varias opciones de una lista de forma independiente**.
- ▶ **Un QCheckBox puede estar en dos estados principales: seleccionado** (con una **marca = 2**) o **deseleccionado (vacío = 0)**, aunque **tenemos** un tercer estado, que indica que una **casilla no está estrictamente ni marcada ni desmarcada (neutro = 1)**.



5.4- QCheckbox (Casillas de verificación)

- ▶ Aunque en este apartado vemos una única casilla, en general, **solemos encontrar varias casillas agrupadas.**

```
1 from PySide6.QtWidgets import QApplication, QMainWindow, QCheckBox
2 import sys
3
4 class MainWindow(QMainWindow):
5     def __init__(self):
6         super().__init__()
7
8         # creamos una casilla y la establecemos de widget central
9         casilla = QCheckBox("Casilla de verificación")
10        self.setCentralWidget(casilla)
11
12 if __name__ == "__main__":
13     app = QApplication(sys.argv)
14     window = MainWindow()
15     window.show()
16     sys.exit(app.exec_())
```

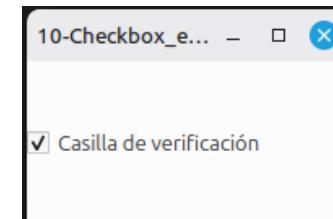


Código [9-Checkbox.py](#)

5.4- QCheckbox (Casillas de verificación)

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

```
# señal para detectar cambios en la casilla  
casilla.stateChanged.connect(self.estado_cambiado)  
  
def estado_cambiado(self, estado):  
    print(estado)
```



```
○ (ModuloDI) alu@Pc-IAW:~/DI_UD2/ModuloDI$  
/home/alu/DI_UD2/ModuloDI/Apartado5/10-C  
sys.exit(app.exec_())  
2
```

Como salida por pantalla tenemos el estado 2, significa que la casilla esta marcada

Código [10-Checkbox_estado1.py](#)

5.4- QCheckbox (Casillas de verificación)

- ▶ Los estados numéricos de la casilla son **0 (desmarcada)** y **2 (marcada)**. Podemos utilizar banderas para analizar de forma más amigable la casilla:

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

# Función con código correcto que muestra el estado de la casilla
def estado_cambiado(self, estado):
    if estado == 2:
        print("Casilla marcada")
    elif estado == 0:
        print("Casilla desmarcada")
    else:
        print("Casilla neutra")
```



```
(ModuloDI) alu@Pc-IAW:~/DI_UD2/ModuloDI$ /home/alu/DI_UD2/ModuloDI/Apartado5/11-Cl
sys.exit(app.exec_())
Casilla marcada
Casilla desmarcada
Casilla marcada
Casilla desmarcada
Casilla marcada
```

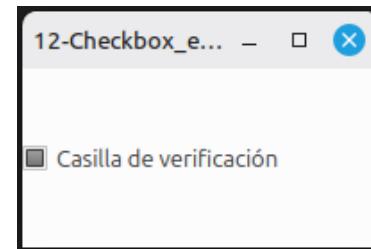
Como salida por pantalla tenemos varios estados, después de pulsar varias veces sobre la casilla

Código [11-Checkbox_estado2.py](#)

5.4- Checkbox (Casillas de verificación)

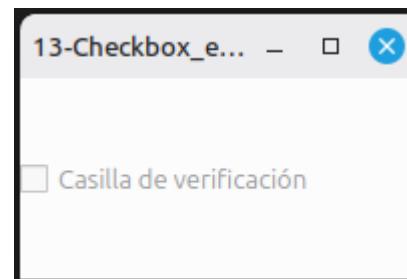
- ▶ Los estados numéricos de la casilla son **0 (desmarcada) y 2 (marcada)**. Podemos utilizar el **estado 1 (tri-estado)** que indica que una casilla no está estrictamente ni marcada ni desmarcada, sino en estado neutro:

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



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

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



Código [12-Checkbox_estado3.py](#)
[13-Checkbox_estado4.py](#)

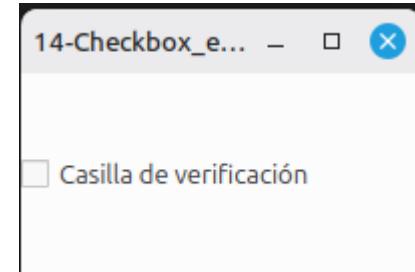
5.4- QCheckbox (Casillas de verificación)

- ▶ Para **consultar el estado actual** de la casilla simplemente utilizaríamos ***isChecked*** o ***isTristate*** para especificar un estado neutro:

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

# establecemos el estado deseado antes de ejecutar el código
casilla.setChecked(0)

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



El método `setChecked`, solo permite un booleano, activo o desactivo, así que, si el triestado esta activo, dará activo si el botón esta activo.

Código [14-Checkbox_estado5.py](#)

5.4- QCheckbox (Casillas de verificación)

- ▶ Para finalizar, **vemos un ejemplo de multicasilla**, pero se puede comprobar que **se necesita la utilización de un contenedor (Layout) y la agrupación de las mismas**. Se verá más adelante.

```
layout = QVBoxLayout()

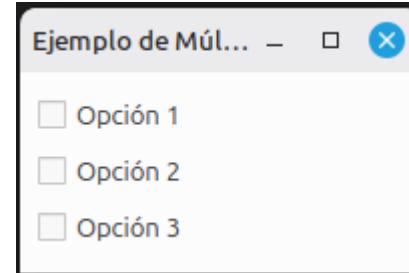
# 1. Crear las casillas
self.check1 = QCheckBox("Opción 1")
self.check2 = QCheckBox("Opción 2")
self.check3 = QCheckBox("Opción 3")

# 2. agrupamos casillas
casillas = [self.check1, self.check2, self.check3]

# 4. Conectar las señales
for check in casillas:
    check.stateChanged.connect(self.on_state_changed)

# 3. Organizar en la interfaz
layout.addWidget(self.check1)
layout.addWidget(self.check2)
layout.addWidget(self.check3)

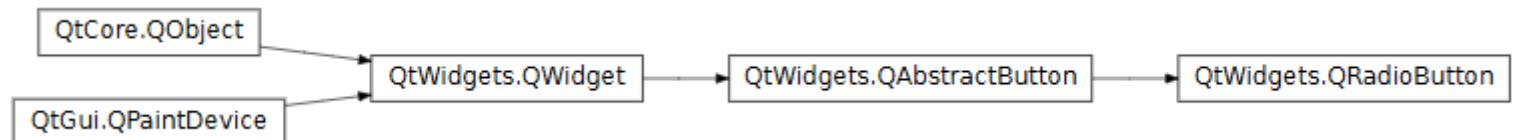
widget = QWidget()
widget.setLayout(layout)
self.setCentralWidget(widget)
```



Código [15-Checkbox_multi.py](#)

5.5- QPushButton (Botones radiales)

- Muy parecidas a las casillas de verificación, pero **con los botones radiales y solo puede marcarse o desmarcarse uno de ellos**, no tienen estado neutro.



- Vemos un pequeño código para la **creación del QPushButton** y la captura de las señales que provoca.

```
class MainWindow(QMainWindow):  
    def __init__(self):  
        super().__init__()  
  
        # creamos un botón radial y lo establecemos de widget central  
        radial = QRadioButton("Botón radial")  
        self.setCentralWidget(radial)  
  
        # señal para detectar cambios en el botón  
        radial.toggled.connect(self.estado_cambiado)
```



Código [16-RadioButton.py](#)

5.6- QComboBox (Desplegable)

- ▶ Los *desplegables* son *listas de opciones de las cuales se pueden seleccionar una única opción*.

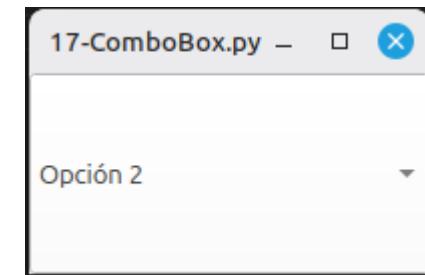


- ▶ Vemos un pequeño código para la creación del QComboBox y la captura de las señales que provoca.

```
# creamos un desplegable
desplegable = QComboBox()
self.setCentralWidget(desplegable)

# añadimos opciones al desplegable, si la primera es vacía, no hay opción
# seleccionada, en otro caso, la primera será la opción por defecto
desplegable.addItems(["", "Opción 1", "Opción 2", "Opción 3"])

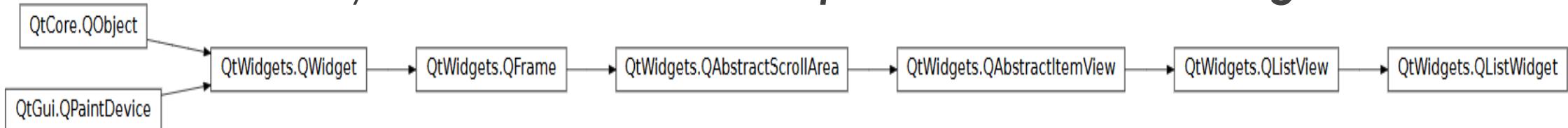
# consultamos el valor actual, índice y texto
print("Índice actual ->", desplegable.currentIndex())
print("Texto actual ->", desplegable.currentText())
```



Código [17-ComboBox.py](#)

5.7- QListWidget (Listas)

- Parecidas al desplegable, pero las opciones no están ocultas ni hay ninguna activa por defecto. En lugar de índices **manejan un tipo de valor llamado QTableWidgetItem y la señal de cambio aquí es currentItemChanged.**

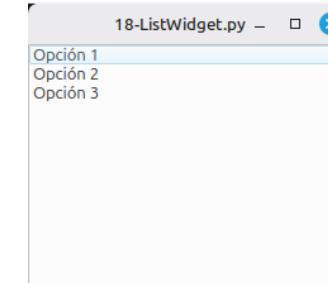


- Vemos un pequeño código para la creación del QListWidget y la captura de las señales que provoca. Para acceder al texto del item , lo hacemos con item.Text(). **Si no hay nada seleccionado, devuelve “None”**

```

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

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



Código [18-ListWidget.py](#)

5.8- QLineEdit (Campo de texto)

- ▶ Los **campos de texto** son los **widgets que permiten capturar contenido escrito por el usuario**.

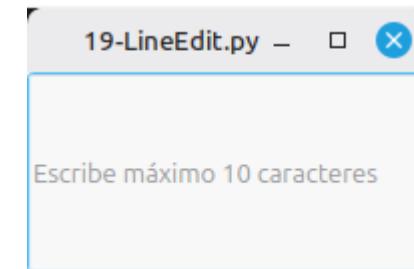


- ▶ Vemos un pequeño código para la creación del QLineEdit y la captura de las señales que provoca.

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

# Probamos algunas opciones
texto.setMaxLength(10)
# Texto inicial (opcional)
texto.setPlaceholderText("Escribe máximo 10 caracteres")

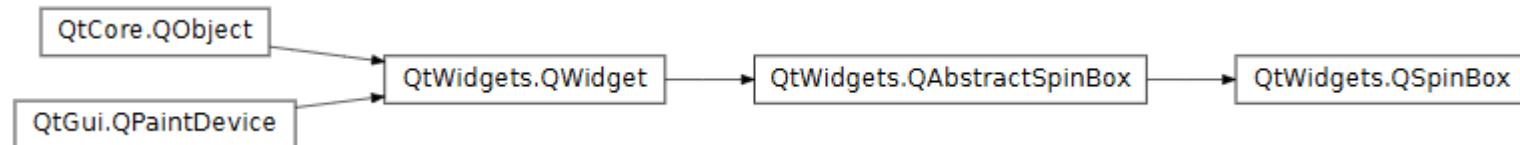
# Probamos algunas señales
texto.textChanged.connect(self.texto_cambiado)
texto.returnPressed.connect(self.enter_presionado)
```



Código [19-LineEdit.py](#)

5.9- QSpinBox (Campo numérico)

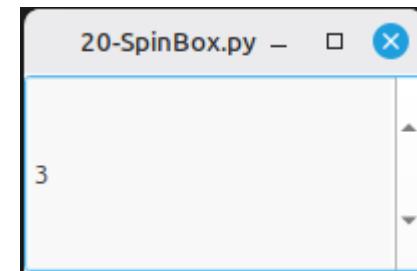
- ▶ Los **campos numéricos fuerzan al usuario a escribir números** y proveen métodos para su control. Veamos primero los enteros.



- ▶ Vemos un pequeño código para la creación del QSpinBox y la captura de las señales que provoca.

```
# creamos un campo numérico entero y le ponemos un valor inicial
numero = QSpinBox()
numero.setValue(3)
self.setCentralWidget(numero)
```

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

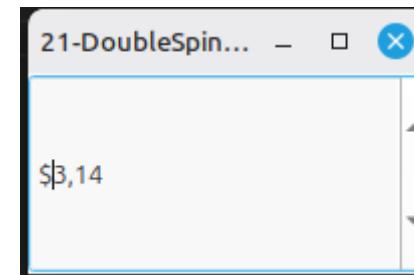


Código [20-SpinBox.py](#)

5.9- QSpinBox (Campo numérico)

- ▶ En cuanto a *los decimales son exactamente igual, pero utilizando el widget QDoubleSpinBox:*
 - QSpinBox -> QDoubleSpinBox
 - numero = QSpinBox() -> numero = QDoubleSpinBox()
 - numero.setSingleStep(1) -> numero.setSingleStep(0.5).

```
# creamos un campo numérico entero y le ponemos un valor inicial
numero = QDoubleSpinBox()
numero.setValue(3.1415)
# Podemos añadir un prefijo o sufijo al número
numero.setPrefix("$")
self.setCentralWidget(numero)
```



Código [21-DoubleSpinBox.py](#)

“

6.- Formas de organización. Uso de Layouts.

”

USO DE LOS LAYOUTS PARA MEJORAR UNA INTERFAZ DE USUARIO.

Creación y configuración de Layouts

6.- Layouts

- ▶ La palabra inglesa “**layout**” se traduce al español como **diseño, plan o disposición**.
- ▶ Se utiliza para hacer referencia a determinado **esquema de disposición y distribución dentro de un diseño**.
- ▶ Este término se utiliza en diferentes áreas como tecnología, diseño gráfico, marketing (mercadotecnia) y arquitectura.
- ▶ El **objetivo** principal es **organizar los componentes de manera funcional y estética, mejorando la experiencia del usuario, optimizando procesos o guiando la mirada**.

6.1 - Widget personalizado

- ▶ Hemos visto ejemplos muy sencillos en una ventana principal con un solo widget, ahora **vamos a implementar varios widgets en el mismo espacio**.
- ▶ Esto significa que **tenemos que organizar varios widgets** y precisamente **para eso existen los layouts**, que se traducirían en español como disposiciones.
- ▶ Utilizaremos diferentes clases como base con fondos coloreados, así veremos exactamente el espacio que ocupan los layouts de una forma muy visual.
- ▶ **A un widget personalizado para visualizar nuestros layouts, lo vamos a llamar caja o contenedor, y lo heredaremos de una simple QLabel.**

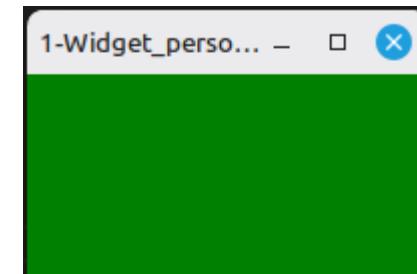
6.1- Widget personalizado

- ▶ **Utilizamos una hoja de estilo para otorgar un color de fondo a nuestra caja mediante el método setStyleSheet.**
- ▶ Vamos a crear una **ventana principal básica** usando esta caja como widget central:

```
# Creamos una subclase de QLabel para personalizar el contenedor
class Contenedor(QLabel):
    def __init__(self, color):
        super().__init__()
        # Establecemos el color de fondo usando hojas de estilo
        self.setStyleSheet(f"background-color:{color}")

# Creamos la ventana principal
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # El contenedor tendrá un color de fondo verde
        caja = Contenedor("green")
        self.setCentralWidget(caja)
```



Código [1-Widget_personal.py](#)

6.2- Layouts básicos

- ▶ Existen **dos tipos de disposición básica** para **organizar elementos vertical u horizontalmente.**
- ▶ Veamos el primer tipo:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QLabel, QVBoxLayout, QWidget
import sys

class Contenedor(QLabel):
    def __init__(self, color):
        super().__init__()
        # Establecemos el color de fondo usando hojas de estilo
        self.setStyleSheet(f"background-color:{color}")

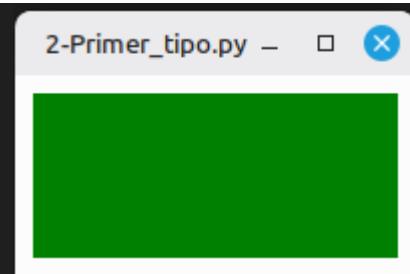
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # Creamos un layout vertical
        layout = QVBoxLayout()
        layout.addWidget(Contenedor("green"))

        # Creamos un dummy widget para hacer de contenedor
        # Esto es necesario porque los layouts no pueden ser asignados directamente
        # a una ventana principal
        widget = QWidget()

        # Le asignamos el layout
        widget.setLayout(layout)

        # Establecemos el dummy widget como widget central
        self.setCentralWidget(widget)
```

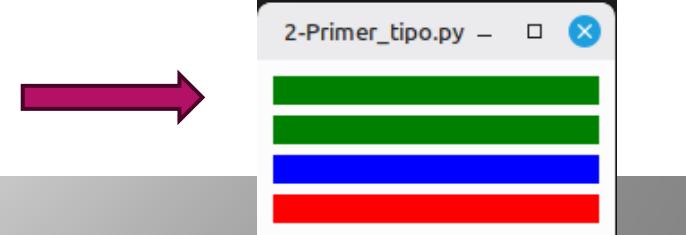


Código [2-Widget_Visual.py](#)

6.2- Layouts básicos

- ▶ **No se permite utilizar un layout como widget central** (pantalla principal). Eso es porque los layouts no son widgets, **no heredan de la clase QWidget**.
- ▶ La forma de manejar esto **es crear un dummy widget para asignarle el layout y usarlo como widget central**.
- ▶ El **layout contiene la caja verde, que a su vez se encuentra dentro del dummy widget asignado como widget principal**. La diferencia más notable es que **un layout tiene espacios y márgenes**, por eso la caja no ocupa todo el espacio.
- ▶ Vamos a **añadir más cajas para ver cómo organiza el espacio automáticamente**:

```
# le añadimos unas cuantas cajas
layout.addWidget(Contenedor("green"))
layout.addWidget(Contenedor("blue"))
layout.addWidget(Contenedor("red"))
```



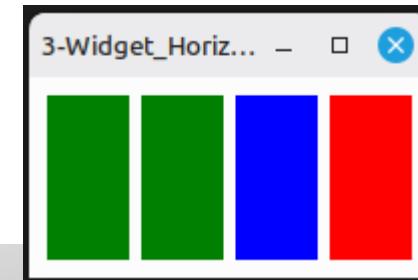
6.2- Layouts básicos

- ▶ El *layout vertical reparte equitativamente el espacio entre los widgets que contiene.*
- ▶ El **segundo tipo**, es cambiar a un *layout horizontal* para ver cómo se reparten los objetos:
 - QVBoxLayout -> QHBoxLayout
 - layout = QVBoxLayout() -> layout = QHBoxLayout()
- ▶ Es exactamente lo mismo, pero en esta ocasión todo se organiza horizontalmente.

```
from PySide6.QtWidgets import QApplication, QMainWindow, QLabel, QHBoxLayout, QWidget
import sys
```

```
# Creamos un layout vertical
layout = QHBoxLayout()
layout.addWidget(Contenedor("green"))

# le añadimos unas cuantas cajas
layout.addWidget(Contenedor("green"))
layout.addWidget(Contenedor("blue"))
layout.addWidget(Contenedor("red"))
```

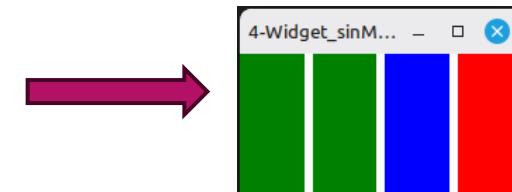


Código [3-Widget Horizontal.py](#)

6.2- Layouts básicos

- ▶ *Podemos modificar los márgenes del layout*, utilizando el método **setContentsMargins** pasándole por orden los píxeles a la izquierda, arriba, derecha y abajo:

```
# modificamos los márgenes  
layout.setContentsMargins(0,0,0,0)
```



- ▶ Y para *quitar el espaciado entre los widgets* utilizaremos **setSpacing** con 0 píxeles:

```
# modificamos el espaciado  
layout.setSpacing(0)
```



- ▶ Nota: estamos usando nuestra caja para visualizar el espacio de cada widget, pero en la vida real estaríamos añadiendo etiquetas, campos de texto y otros widgets para diseñar nuestras vistas,

Código [4-Widget_sinMargenes.py](#)

6.3- Layouts anidados

- ▶ **El potencial de los layouts básicos se pone de manifiesto al mezclarlos**, por ejemplo, partiendo de un layout horizontal que a su vez contiene layouts verticales.
- ▶ Esta técnica nos permite dividir el espacio a voluntad, sin embargo, si lo que necesitamos es un diseño en cuadrícula es mejor utilizar otras disposiciones que veremos más adelante

6.3- Layouts anidados

► Ejemplo de layouts anidados

```
# creamos diferentes layouts para mezclar
layoutHor = QHBoxLayout()
layoutVer1 = QVBoxLayout()
layoutVer2 = QVBoxLayout()

# añadimos una caja al principio del layout 1
layoutHor.addWidget(Contenedor("green"))
# luego anidamos dos layouts verticales
layoutHor.addLayout(layoutVer1)
layoutHor.addLayout(layoutVer2)

# en el primer layout vertical añadimos dos cajas
layoutVer1.addWidget(Contenedor("blue"))
layoutVer1.addWidget(Contenedor("red"))

# en el segundo layout vertical añadimos tres cajas
layoutVer2.addWidget(Contenedor("orange"))
layoutVer2.addWidget(Contenedor("magenta"))
layoutVer2.addWidget(Contenedor("purple"))

# creamos el widget dummy y le asignamos el layout horizontal
widget = QWidget()
widget.setLayout(layoutHor)
```



Código [5-Layout anidados.py](#)

6.4- Layout en cuadrícula

- ▶ *El layout en cuadrícula se basa en crear un único layout compuesto de filas y columnas.*
- ▶ Primero **se crea la cuadrícula y luego se rellena cada hueco** o celda haciendo referencia a ella con índices que empiezan por cero.
- ▶ El tamaño de la cuadrícula vendrá determinado automáticamente por los mayores índices con un widget, lo que generará huecos vacíos si no los rellenamos explícitamente.

6.4- Layouts en cuadricula

- ▶ **Utilizamos la librería y el layout QGridLayout**, veamos un ejemplo sin todas las celdas de colores.

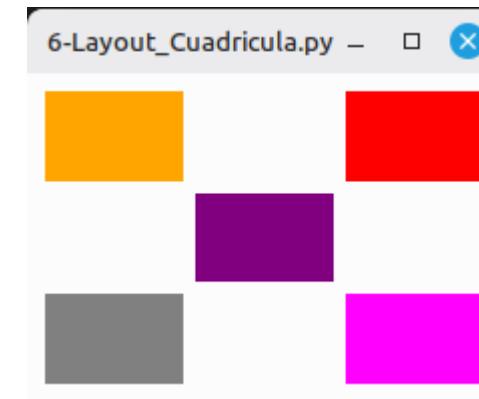
```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QLabel, QGridLayout, QWidget)
import sys

# creamos un layout en cuadrícula
cuadricula = QGridLayout()

# añadimos widgets en las celdas usando los índices
cuadricula.addWidget(Contenedor("orange"), 0, 0)
cuadricula.addWidget(Contenedor("purple"), 1, 1)
cuadricula.addWidget(Contenedor("magenta"), 2, 2)
cuadricula.addWidget(Contenedor("gray"), 2, 0)
cuadricula.addWidget(Contenedor("red"), 0, 2)

# creamos el widget dummy y le asignamos el layout horizontal
widget = QWidget()
widget.setLayout(cuadricula)

self.setCentralWidget(widget)
```



Código [6-Layout Cuadricula.py](#)

6.4- Layouts en cuadricula

- ▶ Para **generar dinámicamente una cuadrícula** con cajas de colores aleatorios, **utilizamos dos bucles for**.

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

        # creamos un layout en cuadricula
        cuadricula = QGridLayout()

        # bucles for para generar una cuadrícula
        for fila in range(5):
            for columna in range(5):
                # añadimos una caja de color aleatorio
                color = str(hex(random.randint(0, 16777215))) # int(0xFFFFFFFF)
                print("[",fila,"][", columna,"]",color, "-->", f"#{color[2:]}")
                cuadricula.addWidget(Contenedor(f"#{color[2:]})", fila, columna)

        # creamos el widget dummy y le asignamos el layout horizontal
        widget = QWidget()
        widget.setLayout(cuadricula)

        self.setCentralWidget(widget)
```



Existe un **ERROR**
en este código

Código [7-Cuadricula_error.py](#)

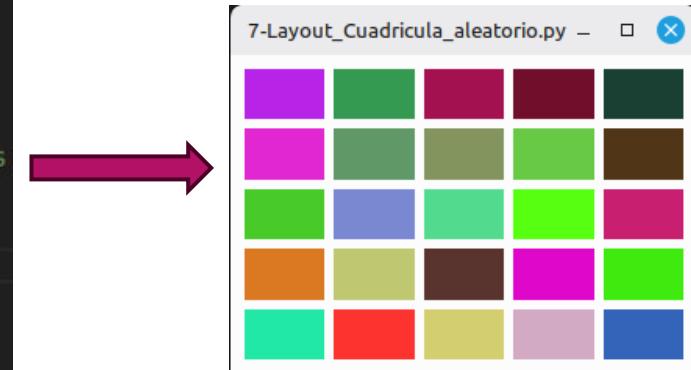
6.4- Layouts en cuadricula

- ▶ Veamos por qué **existe un error en el código anterior**. Ya que no se percibe ningún warning ni error grave. Se debe a que **muchas celdas se ven de color blanco y con códigos de color no existente**.
- ▶ Se debe a que **los códigos hexadecimales para los colores se basan en 6 dígitos**, y al calcular un entero random, podemos obtener valores que, al convertir en hexadecimal, no nos aporte esos 6 dígitos.
- ▶ Se recomienda completar esos dígitos para que no exista el error

```
# bucles for para generar una cuadrícula
for fila in range(5):
    for columna in range(5):
        # añadimos una caja de color aleatorio de 6 digitos
        # intento es una cadena que empieza con 0x y tiene entre 1 y 6 digitos hexadecimales
        intento = str(hex(random.randint(0, 16777215))) # int(0xFFFFFFF)

        # le quitamos a la cadena el 0x y nos aseguramos que tenga 6 digitos
        # añadiendo ceros a la izquierda si es necesario
        color=intento[2: ].zfill(6)

        print("[",fila,"][", columna,"]",color, "-->", f"#{color}")
        cuadricula.addWidget(Contenedor(f"#{color}"), fila, columna)
```



Código 7-1-Cuadricula sinerror.py

6.4- Layouts en cuadricula

- ▶ Este tipo de Layout, nos **permite que utilicemos varios huecos para colocar los componentes que deseamos**. Solo tenemos que escribirlo de forma correcta a la hora de insertar nuestros componentes.
- ▶ Vemos los valores que indican posiciones y tamaños.

The diagram shows a code snippet for adding a widget to a grid layout:

```
self.grid.addWidget(self.num1, 1, 1, 1, 1, Qt.AlignCenter)
```

Annotations with green arrows point to specific parts of the code:

- A vertical arrow points to the first parameter `self.num1`, labeled "WIDGET".
- A horizontal arrow points to the second parameter `1`, labeled "FILA".
- A horizontal arrow points to the third parameter `1`, labeled "CTD-FILAS".
- A horizontal arrow points to the fourth parameter `1`, labeled "ALINEACIÓN".
- A vertical arrow points to the fifth parameter `1`, labeled "COLUMN".
- A horizontal arrow points to the sixth parameter `1`, labeled "CTD-COLUMNS".

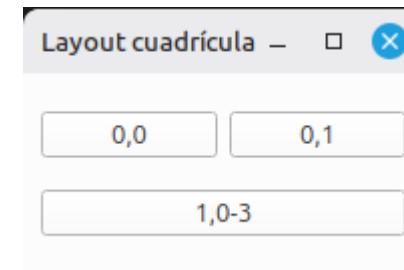
6.4- Layouts en cuadricula

- Veamos un ejemplo:

```
# Creamos un objeto layout cuadrícula
layout_cuadrícula = QGridLayout()
componente_principal = QWidget()
componente_principal.setLayout(layout_cuadrícula)
self.setCentralWidget(componente_principal)

# Añadimos cuatro botones a la primera fila
layout_cuadrícula.addWidget(QPushButton('0,0'), 0, 0)
layout_cuadrícula.addWidget(QPushButton('0,1'), 0, 1)

# Añadimos un botón a la segunda fila que ocupe dos columnas
layout_cuadrícula.addWidget(QPushButton('1,0-3'), 1, 0, 1, 2)
```



6.5- Layout en formulario

- ▶ Si lo que necesitamos es ***una estructura para manejar un formulario*** podemos usar un ***QFormLayout*** que ***nos permite añadir etiquetas y widgets en fila de una forma más cómoda*** que las cuadriculas.
- ▶ Dependiendo del sistema operativo el formulario se visualizará de forma diferente con el objetivo de respetar la integración, pero es posible cambiar la alineación de las etiquetas y los widgets manualmente.

6.5- Layout en formulario

- ▶ Ejemplo de este layout en formulario.

```
class Contenedor(QLabel):
    def __init__(self, color):
        super().__init__()
        self.setStyleSheet(f"background-color:{color}")

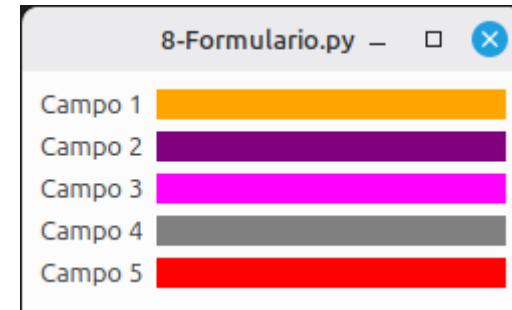
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # creamos un layout en formulario
        formulario = QFormLayout()

        # añadimos widgets con etiquetas en filas
        formulario.addRow("Campo 1", Contenedor("orange"))
        formulario.addRow("Campo 2", Contenedor("purple"))
        formulario.addRow("Campo 3", Contenedor("magenta"))
        formulario.addRow("Campo 4", Contenedor("gray"))
        formulario.addRow("Campo 5", Contenedor("red"))

        # creamos el widget dummy y le asignamos el layout
        widget = QWidget()
        widget.setLayout(formulario)

        self.setCentralWidget(widget)
```



Código [8-Formulario.py](#)

6.6- Layout apilado

- ▶ *Es una clase de Qt que organiza un grupo de widgets en una "pila", mostrando solo uno a la vez.* Se crea con la instrucción: **QStackedLayout**
- ▶ *Es similar a las pestañas de un navegador,* donde cada "pestaña" (o widget) se muestra al hacer clic en un control (por ejemplo: un botón).
- ▶ *Esta disposición es útil para crear interfaces de usuario donde se quiere presentar diferentes vistas* o formularios en un mismo espacio.

6.6- Layout apilado

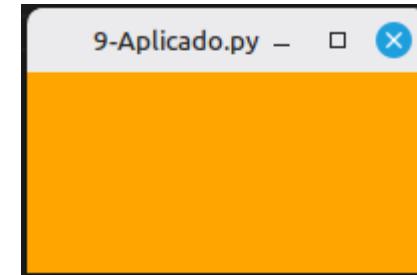
- ▶ Ejemplo de este layout apilado. En el código de prueba se encuentra como cambiar de widget utilizando las flechas izquierda y derecha del teclado.
- ▶ Hemos introducido los eventos, pero podríamos haber utilizado unos botones para cambiar de índice sin problema.

```
# creamos un layout apilado
layout = QStackedLayout()

# Añadimos varios widgets unos sobre otros
layout.addWidget(Caja("orange"))
layout.addWidget(Caja("magenta"))
layout.addWidget(Caja("purple"))
layout.addWidget(Caja("red"))

# creamos el widget dummy y le asignamos el layout apilado
widget = QWidget()
widget.setLayout(layout)

self.setCentralWidget(widget)
```



Código [9-Aplicado.py](#)

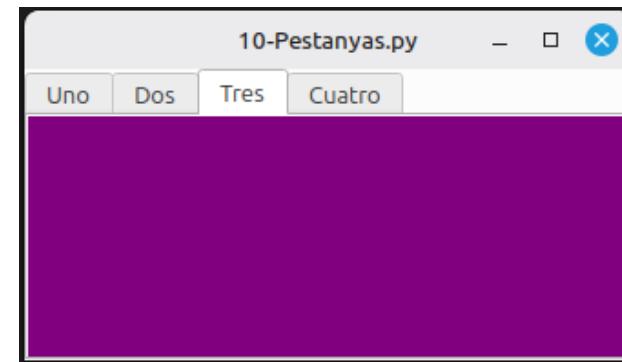
6.7- Layout con pestañas

- ▶ Esta disposición que veremos es **con pestañas utilizando un QTabWidget**.
- ▶ Se trata de una **variante del apilado con un control más visual**. Esta variante **sí hereda de la clase QWidget** y por tanto **no requiere un dummy widget**:

```
# creamos un layout de pestañas
tabs = QTabWidget()

# Añadimos varios widgets como pestañas con nombres
tabs.addTab(Caja("orange"), "Uno")
tabs.addTab(Caja("magenta"), "Dos")
tabs.addTab(Caja("purple"), "Tres")
tabs.addTab(Caja("red"), "Cuatro")

# asignamos las pestañas como widget central
self.setCentralWidget(tabs)
```



Código [10-Pestanyas.py](#)

6.7- Layout con pestañas

- ▶ Algunas opciones interesantes de este widget es que **podemos modificar la posición de las pestañas**:

```
# configuración de los tabs de las pestanas en la orientación deseada  
# West, East, North, South  
tabs.setTabPosition(QTabWidget.West)
```

- ▶ O hacer que las pestañas se puedan arrastrar para cambiar el orden:

```
# permitimos que las pestanas se puedan mover arrastrándolas  
tabs.setMovable(True)
```

6.8- Estiramiento y políticas de medida

- ▶ Ahora **vayamos a controlar como se adaptan los widgets cuando la ventana cambia de medida**: como **repartir el espacio** entre elementos, y como hacer que unos crecen y otros permanecen fijos.

Estiramientos (addStretch())

- ▶ Los **estiramientos (stretches)** se utilizan para añadir **espacios flexibles** dentro de un *Layout*. Son **como “muelles” invisibles** que **ocupan todo el espacio disponible y empujan los widgets**.

6.8- Estiramiento y políticas de medida

► Veamos algunos ejemplos:

- Ejemplo básico, donde los elementos quedan uno arriba y el otro abajo y el espacio flexible queda entre ellos

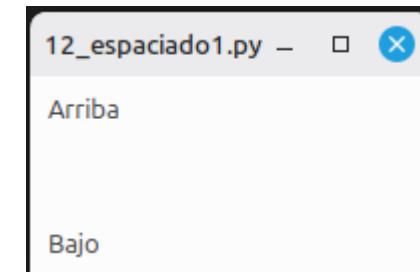
```
# Creamos un layout vertical
layout = QVBoxLayout()

layout.addWidget(QLabel("Arriba"))
layout.addStretch() # Espacio flexible
layout.addWidget(QLabel("Bajo"))

# Creamos un dummy widget para hacer de contenedor
# Esto es necesario porque los layouts no pueden ser asignados directamente
# a una ventana principal
widget = QWidget()

# Le asignamos el layout
widget.setLayout(layout)

# Establecemos el dummy widget como widget central
self.setCentralWidget(widget)
```



Código [12_espaciado1.py](#)

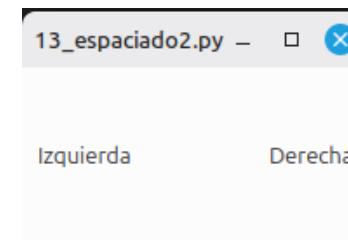
6.8- Estiramiento y políticas de medida

- ▶ Veamos algunos ejemplos:

- **Ejemplo horizontal**, donde los elementos quedan a derecha e izquierda y el espacio flexible queda entre ellos

```
# Creamos un layout vertical
layout = QHBoxLayout()

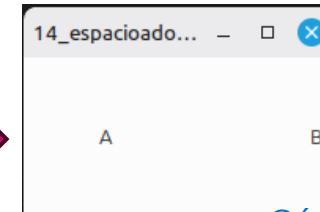
layout.addWidget(QLabel("Izquierda"))
layout.addStretch() # Espacio flexible
layout.addWidget(QLabel("Derecha"))
```



- Ejemplo **factor de estiramiento**, donde podemos indicar un valor (**addStretch(factor)**) para repartir el espacio entre varios “muelles”. En este ejemplo el espacio entre A y B será el triple que el dejado al principio de la ventana.

```
# Creamos un layout vertical
layout = QHBoxLayout()

layout.addStretch(1)
layout.addWidget(QLabel("A"))
layout.addStretch(3)
layout.addWidget(QLabel("B"))
```



Código [13_espaciado2.py](#)
[14_espaciado3.py](#)

6.8- Estiramiento y políticas de medida

Políticas de medida (QSizePolicy)

- ▶ **Cada widget tiene una política de medida que define como se comporta dentro del layout.** Esto permite controlar si el widget puede crecer, reducirse o permanecer fijo.
- ▶ Tipos más comunes:

Política	Descripción
Fixed	Medida fija. No crece ni se reduce.
Preferred	Medida recomendada (crece solo si hay espacio).
Expanding	Se expande para llenar el espacio disponible.
Minimum	Ocupa la medida mínima necesaria.

6.8- Estiramiento y políticas de medida

- **Veamos un ejemplo.** Tenemos un botón fijo y otro que cambia según redimensionamos la ventana:

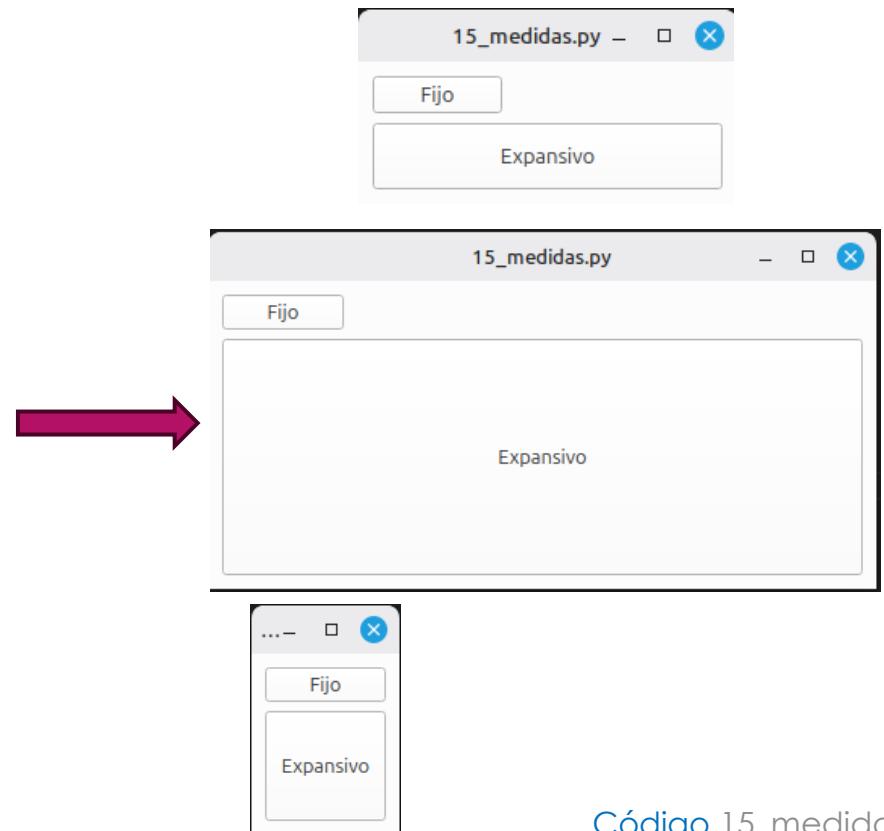
```
# Creamos un layout vertical
layout = QVBoxLayout()

boton1 = QPushButton("Fijo")
boton1.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
boton2 = QPushButton("Expansivo")
boton2.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

layout.addWidget(boton1)
layout.addWidget(boton2)

# Creamos un dummy widget para hacer de contenedor
# Esto es necesario porque los layouts no pueden ser asignados directamente
# a una ventana principal
widget = QWidget()

# Le asignamos el layout
widget.setLayout(layout)
```



Código [15_medidas.py](#)

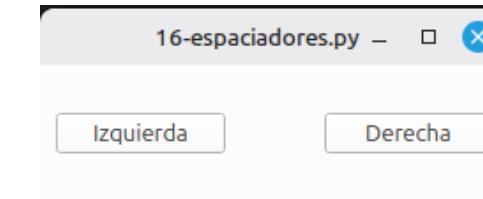
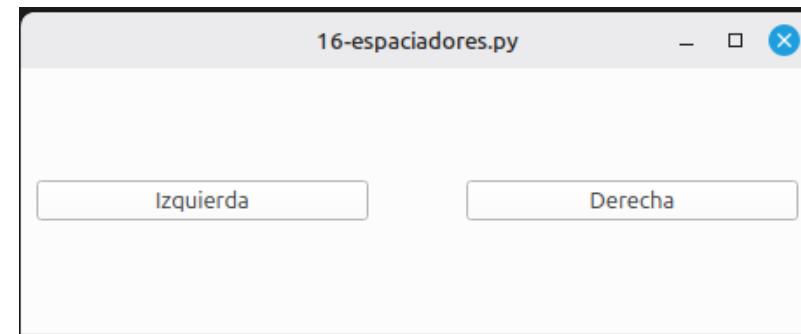
6.8- Estiramiento y políticas de medida

Espaciadores (addSpacing(px))

- ▶ Los espaciadores son zonas vacías que controlan las separaciones entre widgets.
- ▶ Ejemplo, entre los botones siempre habrá 50 pixeles de separación, por mucho que redimensionemos la ventana:

```
# Creamos un layout vertical
layout = QHBoxLayout()

layout.addWidget(QPushButton("Izquierda"))
layout.addSpacing(50)
layout.addWidget(QPushButton("Derecha"))
```



Código [16-espaciadores.py](#)

6.8- Estiramiento y políticas de medida

Alineación de Widgets

- ▶ Cuando añades un widget a un layout, puedes *indicar la alineación exacta dentro de su espacio*:

```
layout.addWidget(boton1, alignment=Qt.AlignRight | Qt.AlignVCenter)
```

Constante Qt	Significado
Qt.AlignLeft	Alinea a la izquierda
Qt.AlignRight	Alinea a la derecha
Qt.AlignTop	Se Alinea arriba
Qt.AlignBottom	Alinea abajo
Qt.AlignHCenter	Centra horizontalmente
Qt.AlignVCenter	Centra verticalmente
Qt.AlignCenter	Centra totalmente (H + V)

6.9- Ejemplo múltiples Layouts

- ▶ Vamos a realizar la combinación de múltiples Layouts.
- ▶ Vamos a ver si somos capaces de desarrollar esta estructura:



Código [11-Ejemplo.py](#)

6.9- Ejemplo múltiples Layouts

- Una posibilidad:

```

1  from PySide6.QtWidgets import (QApplication, QMainWindow,
2 | | | QWidget, QHBoxLayout, QVBoxLayout, QLabel)
3  import sys
4
5  class Contenedor(QLabel):
6      def __init__(self,color):
7          super().__init__()
8          self.setStyleSheet(f"background-color:{color}")
9
10 class MainWindow(QMainWindow):
11     def __init__(self):
12         super().__init__()
13         self.setWindowTitle("Ejemplo Layouts Multiples")
14
15         # Creación de layouts, uno horizontal (1) y dos verticales (2 y 3)
16         layout1=QHBoxLayout()
17         layout2=QVBoxLayout()
18         layout3=QVBoxLayout()
19
20         # Añadimos widgets al layout2
21         layout2.addWidget(Contenedor("red"))
22         layout2.addWidget(Contenedor("yellow"))
23         layout2.addWidget(Contenedor("purple"))
24
25         # Incluimos layouts2 dentro del Layout1, es importante el orden
26         layout1.addLayout(layout2)

```

```

28         # Añadimos widgets al layout1
29         layout1.addWidget(Contenedor("green"))
30
31         # Añadimos widgets al layout3
32         layout3.addWidget(Contenedor("red"))
33         layout3.addWidget(Contenedor("purple"))
34
35         # Incluimos layouts3 dentro del Layout1,
36         layout1.addLayout(layout3)
37
38         #Quitar margenes o poner margenes
39         layout1.setContentsMargins(5,5,5,5)
40         layout2.setContentsMargins(0,0,0,0)
41         layout3.setContentsMargins(0,0,0,0)
42
43         #Quitar o poner margenes externos
44         layout1.setSpacing(5)
45         layout2.setSpacing(5)
46         layout3.setSpacing(5)
47
48         # Un Layout no se puede mostrar como Widget central
49         # asi que lo metemos en un widget dummy
50         widget=QWidget()
51         widget.setLayout(layout1)
52         self.setCentralWidget(widget)
53
54
55     if __name__ == "__main__":
56         app = QApplication(sys.argv)
57         window = MainWindow()
58         window.show()
59         sys.exit(app.exec_())

```

“

7.- Cuadros de diálogo, predeterminados o personalizados.

”

CREAMOS CUADROS DE DIÁLOGOS DE DIFERENTES MENSAJES

Creación de cuadros de diálogo

7- Cuadros de Diálogo

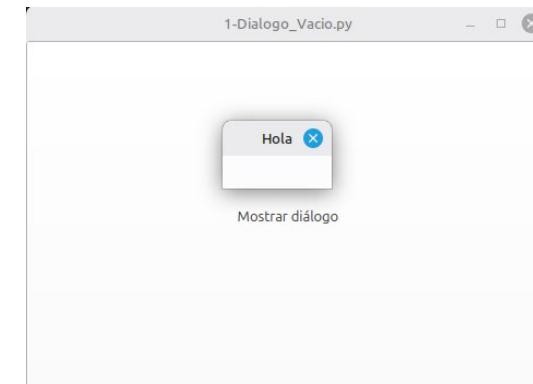
- ▶ Los cuadros de diálogo son elementos gráficos de las interfaces de usuario que sirven para recopilar información, mostrar mensajes o confirmar acciones con el usuario.
- ▶ Son ventanas secundarias que se abren para interactuar con el software y pueden contener campos de texto, menús desplegables y botones para ejecutar o cancelar tareas.
- ▶ Existen dos tipos principales: los modales, que bloquean la interacción con la ventana principal hasta que se cierran, y los no modales, que permiten continuar trabajando en otras partes de la aplicación.
- ▶ Todos los diálogos parten de la clase QDialog y están pensados para mostrar o pedir información al usuario en una nueva ventana temporal.

7.1- Diálogos personalizados

- ▶ Al estar pensados para aparecer de forma emergente, los diálogos requieren una acción desencadenante para aparecer.
- ▶ Este dialogo es modal, no permite seguir con la ejecución de la aplicación hasta que no se cierre el diálogo.
- ▶ Vamos a partir de un diálogo vacío para ver su funcionamiento, tenemos una ventana principal sobre la que pinchamos para que aparezca el diálogo:

```
boton = QPushButton("Mostrar diálogo")
boton.clicked.connect(self.boton_clicado)
self.setCentralWidget(boton)

def boton_clicado(self):
    dialogo = QDialog(self)
    dialogo.setWindowTitle("Hola")
    dialogo.exec_()
```



7.1- Diálogos personalizados

- ▶ El diálogo QDialog está pensado para extenderlo, vamos a personalizar algunas opciones creando nuestro propio diálogo heredando de esta clase:

```
class Dialogo(QDialog):
    def __init__(self):
        super().__init__()
        self.resize(240, 120)
        self.setWindowTitle("Hola")

        # creamos un layout y lo establecemos en el widget
        layout = QVBoxLayout()
        self.setLayout(layout)

        # podemos añadir una etiqueta
        layout.addWidget(QLabel("Diálogo de prueba"))

        # creamos unos botones predeterminados
        botones = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel)

        # y los añadimos al layout
        layout.addWidget(botones)

        # configuramos unas señales predeterminadas
        botones.accepted.connect(self.accept)
        botones.rejected.connect(self.reject)
```

```
class Dialogo(QDialog):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Hola")

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

    def boton_clicado(self):
        dialogo = Dialogo()
        dialogo.exec_()
```

- ▶ Lo más esencial que nos permite un diálogo es interactuar con él a través de botones. Estos botones se configuran en su propia caja de botones llamada QDialogButtonBox que normalmente pondremos dentro de un layout donde organizar el espacio:

7.1- Diálogos personalizados

- ▶ Si deseamos que un diálogo se superponga sobre la ventana donde se ha creado hay que pasarle la instancia del padre al crearlo:

```
class Dialogo(QDialog):  
    def __init__(self, parent=None): # editado  
        super().__init__(parent) # editado
```

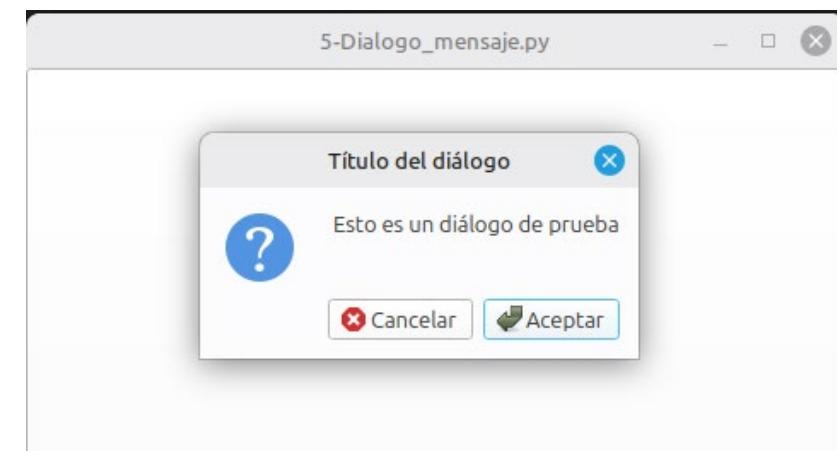
- ▶ Por cierto, los botones del diálogo se encuentran localizados, es decir que permiten traducciones. El problema es que hay que dar de alta un traductor y para dos botones no compensa hacerlo. Es más fácil cambiar el texto a mano:

```
# traducción en tiempo real de los botones  
botones.button(QDialogButtonBox.Ok).setText("Aceptar")  
botones.button(QDialogButtonBox.Cancel).setText("Cancelar")
```

7.2- Diálogos de mensaje

- ▶ Si lo que deseamos en enviar un mensaje al usuario tenemos a nuestra disposición una clase llamada QMessageBox que simplifica la personalización de un QDialog. Básicamente podemos modificar los atributos sin crear nuestra propia clase heredada y además podemos usar iconos predeterminados:

```
def boton_clicado(self):  
    # creamos un diálogo de mensaje con un título y un texto  
    dialogo = QMessageBox(self)  
    dialogo.setWindowTitle("Título del diálogo")  
    dialogo.setText("Esto es un diálogo de prueba")  
    # añadimos unos botones y los traducimos  
    dialogo.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)  
    dialogo.button(QMessageBox.Ok).setText("Aceptar")  
    dialogo.button(QMessageBox.Cancel).setText("Cancelar")  
    # configuramos un ícono  
    dialogo.setIcon(QMessageBox.Question)  
  
    # ejecutamos el diálogo y capturamos la respuesta  
    respuesta = dialogo.exec_()  
    # ahora debemos comprobar qué tipo de botón se ha clicado  
    if respuesta == QMessageBox.Ok:  
        print("Diálogo aceptado")  
    else:  
        print("Diálogo denegado")
```



7.3- Diálogos predeterminados

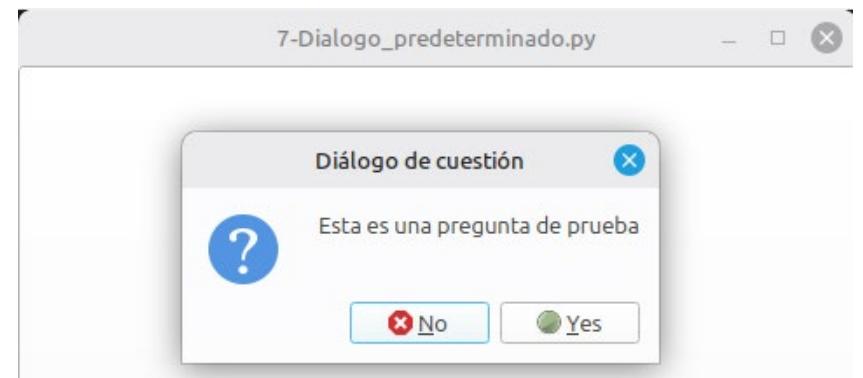
- ▶ Por suerte no necesitamos crear diálogos todo el tiempo, Qt incluye diálogos predeterminados para realizar diferentes tareas.

Mensaje de cuestión

- ▶ Realiza una pregunta y espera una respuesta “SI” u otra opción, por defecto “NO”

```
def boton_clicado(self):
    # creamos un diálogo de tipo cuestión, "titulo" y "pregunta"
    dialogo = QMessageBox.question(self, "Diálogo de cuestión",
                                    "Esta es una pregunta de prueba")

    # ahora podemos comprobar qué tipo de botón se devuelve
    # hay varios tipos predefinidos, como Yes, No, Cancel, Ok, etc.
    # en este caso no utilizamos else por ese motivo
    print(dialogo)
    if dialogo == QMessageBox.Yes:
        print("Ha respondido sí")
```



7.3- Diálogos personalizados

Mensaje de Acerca de

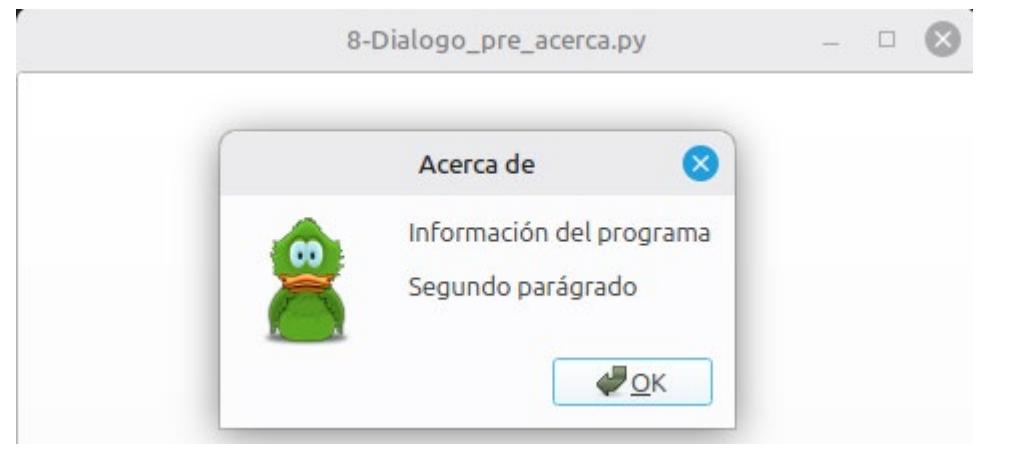
- ▶ Para mostrar información del programa o el autor.
- ▶ Este diálogo toma por defecto el ícono de la ventana, vamos a añadir uno de ejemplo para verlo, usaremos la función para generar rutas absolutas.

```
# obtener el path absoluto de un archivo
def absPath(file):
    #print(str(Path(__file__).parent.absolute() / file))
    return str(Path(__file__).parent.absolute() / file)

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(480, 320)
        self.setWindowIcon(QIcon(absPath("./imagenes/icon.png")))

        boton = QPushButton("Mostrar diálogo")
        boton.clicked.connect(self.boton_clicado)
        self.setCentralWidget(boton)

    def boton_clicado(self):
        dialogo = QMessageBox.about(self, "Acerca de",
                                    "<p>Información del programa</p><p>Segundo párrafo</p>")
        # podemos analizar el tipo de botón clicado para actuar en consecuencia
        print(dialogo)
```

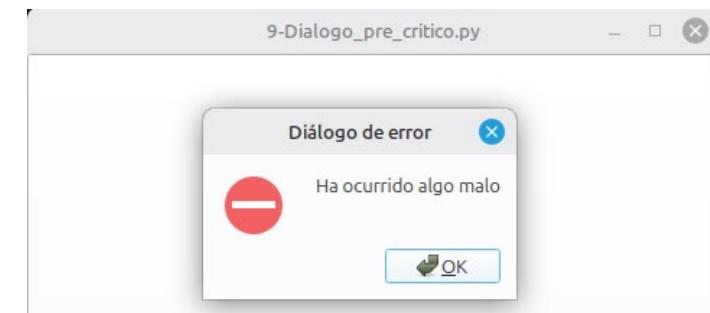


7.3- Diálogos personalizados

Mensaje crítico

- ▶ Este diálogo reproduce un sonido de error mientras muestra la ventana.

```
def boton_clicado(self):
    dialogo = QMessageBox.critical(self, "Diálogo de error",
                                    "Ha ocurrido algo malo")
    print(dialogo)
```

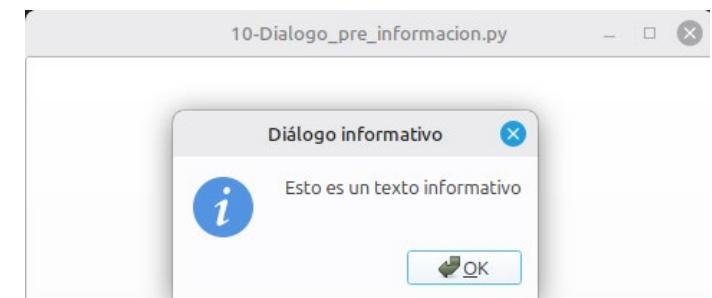


7.3- Diálogos personalizados

Mensaje informativo

- ▶ Para mostrar información genérica.

```
def boton_clicado(self):
    dialogo = QMessageBox.information(self,
                                      "Diálogo informativo", "Esto es un texto informativo")
    print(dialogo)
```

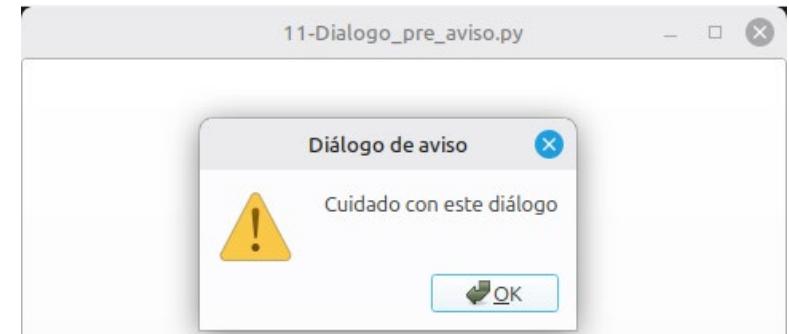


7.3- Diálogos personalizados

Mensaje de aviso

- ▶ Para mostrar un aviso.

```
def boton_clicado(self):  
    dialogo = QMessageBox.warning(self, "Diálogo de aviso",  
                                "Cuidado con este diálogo")  
    print(dialogo)
```



7.4- Diálogos modales y no modales

- ▶ Los diálogos (QDialog) pueden funcionar de dos maneras según si bloquean o no la interacción con la ventana principal:

Tipo	Descripción	Método
Modal	Bloquea el resto de la interfaz hasta que se cierre	exec()
No modal	Permite seguir trabajando con la ventana principal mientras el diálogo está abierto	show()

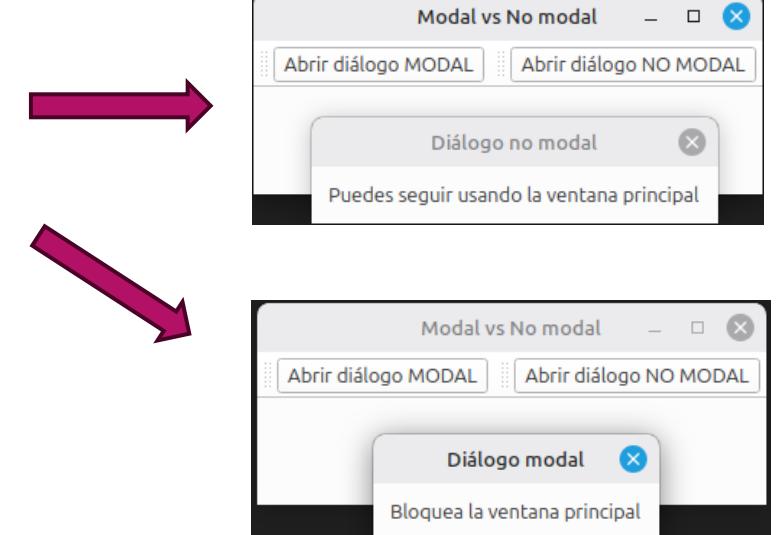
- ▶ El modo modal es útil cuando hace falta una respuesta inmediata del usuario (confirmar, aceptar, etc.). El modo no modal es adecuado para mostrar información o paneles auxiliares que no interrumpen el flujo de trabajo.

7.4- Diálogos modales y no modales

- ▶ Ejemplo de código. Creamos un diálogo personalizado que muestre un texto y según como lo ejecutemos, será modal o no modal.

```
# Método para abrir un diálogo modal
def abre_modal(self):
    dlg = InfoDialog("Diálogo modal", "Bloquea la ventana principal", self)
    dlg.exec() # Modal → bloquea hasta que se cierre

# Método para abrir un diálogo no modal
def abre_no_modal(self):
    self.dlg = InfoDialog("Diálogo no modal", "Puedes seguir usando la ventana principal", self)
    self.dlg.show() # No modal → no bloquea
```



7.5- Activando las traducciones

- ▶ Traducir los diálogos predeterminados a mano hace que se pierda su simplicidad, por eso es recomendable activar las traducciones. Esto implica que hay que distribuir los ficheros de traducción junto a los ejecutables, pero eso es algo que veremos más adelante.
- ▶ Para activar la localización del idioma del sistema operativo debemos traducir la aplicación de la siguiente forma:

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

    # envolvemos la aplicación con el traductor
    translator = QTranslator(app)
    # recuperamos el directorio de traducciones
    translations = QLibraryInfo.location(QLibraryInfo.TranslationsPath)
    # cargamos la traducción en el traductor
    translator.load("qt_es", translations)
    # la aplicamos
    app.installTranslator(translator)

    # idiomas disponibles en la carpeta de traducciones
    print(translations)

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

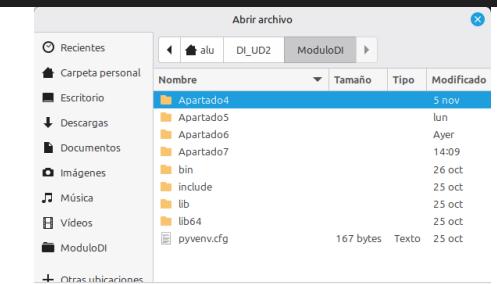
7.6- Diálogos específicos

- ▶ Por último, tenemos diálogos para usos específicos. Es importante tener en cuenta que es necesario activar las traducciones o los textos aparecerán en inglés.

Diálogos de fichero

- ▶ Se utilizan para generar la ruta a un fichero usando el explorador, es decir, no afectan al fichero en sí y solo sirven para saber donde se encuentra un fichero, ya sea para abrirlo o para guardarla. Este modo es muy útil porque si el fichero ya existe te avisa de que se va a sobrescribir.

```
def boton_clicado(self):
    #Abrir un fichero
    fichero, _ = QFileDialog.getOpenFileName(self,
                                              "Abrir archivo", ".")
    print(fichero)
```



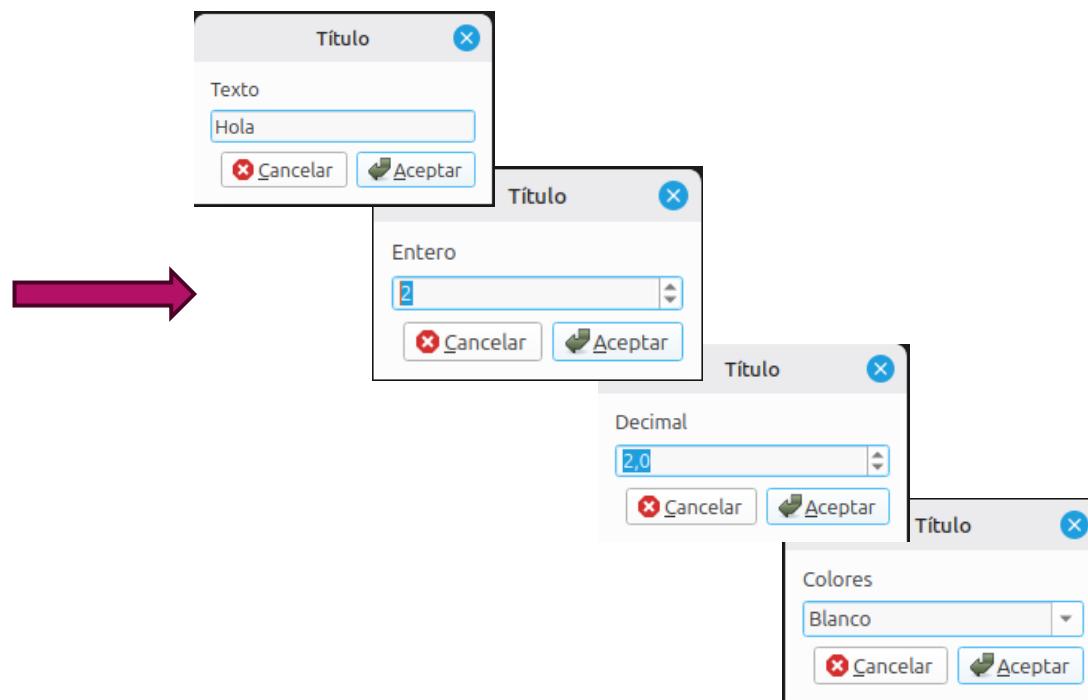
```
def boton_clicado(self):
    #Guardar un fichero
    fichero, _ = QFileDialog.getSaveFileName(self,
                                              "Guardar archivo", ".")
    print(fichero)
```

7.6- Diálogos específicos

Diálogos de entrada de datos

- ▶ Pensados para pedir un dato concreto al usuario, en este ejemplo solicitamos 4 datos.

```
def boton_clicado(self):  
    dialogo = QInputDialog.getText(self, "Título", "Texto")  
    dialogo = QInputDialog.getInt(self, "Título", "Entero")  
    dialogo = QInputDialog.getDouble(self, "Título", "Decimal")  
    dialogo = QInputDialog.getItem(self, "Título",  
|    |    |    "Colores", ["Rojo", "Azul", "Blanco", "Verde"])
```



7.6- Diálogos específicos

- Antes de continuar con otros elementos, veamos algo importante, como capturar los datos de estos diálogos:

```
# Guardo de cada dialogo el valor y el estado de aceptación
# Fijaros que recibe dos parametros de salida, el valor y el estado (aceptar OK/cancelar)
nombre, estado_tit = QInputDialog.getText(self, "Titulo", "Nombre")
edad, estado_eda = QInputDialog.getInt(self, "Título", "Edad")
precio, estado_pre = QInputDialog.getDouble(self, "Título", "PVP")
color, estado_col = QInputDialog.getItem(self, "Título",
                                         "Color", ["Rojo", "Azul", "Blanco", "Verde"])

# si todos los estados de aceptación son válidos, imprimo por pantalla
if (estado_tit and estado_eda and estado_pre and estado_col):
    print ("Nombre: ", nombre, " edad: ", edad, " PVP: ", precio, " Color: ",color)
```

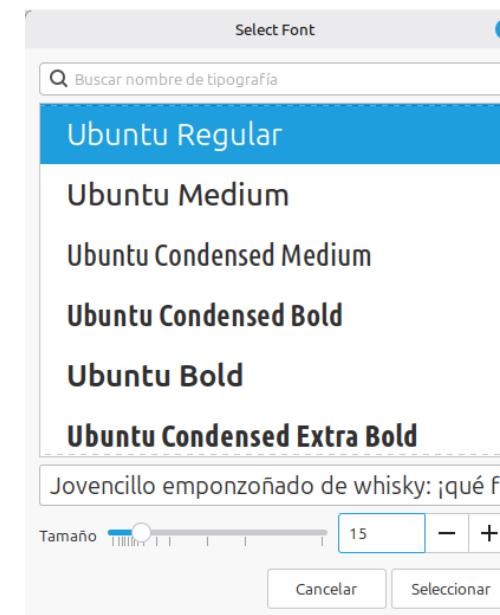
7.6- Diálogos específicos

Diálogos de fuente y color

- ▶ Estos tienen el objetivo de seleccionar fuentes del sistema y colores.

```
# Definición de la acción del botón
def boton_clicado(self):

    # Crea un Diálogo de fuente y recibe dos resultados:
    # Primero si hay una fuente seleccionada y la fuente
    confirmado, fuente = QFontDialog.getFont(self)
    if confirmado:
        # fuente es un objeto QFont
        self.boton.setFont(fuente)
```



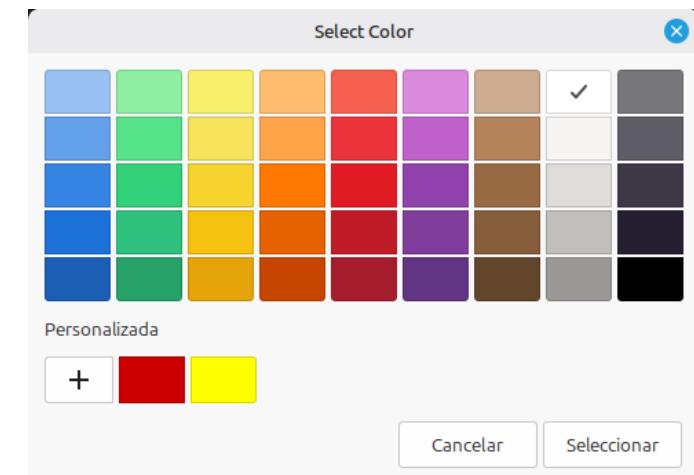
7.6- Diálogos específicos

Diálogos de fuente y color

- ▶ Estos tienen el objetivo de seleccionar fuentes del sistema y colores.

```
# Definición de la acción del botón
def boton_clicado(self):
    # Crea un Diálogo de color y recibe un resultado:
    # el color seleccionado
    color = QColorDialog.getColor()

    if color.isValid():
        # color es un objeto QColor, name() devuelve su código hexadecimal
        self.boton.setStyleSheet(f"background-color: {color.name()}")
```



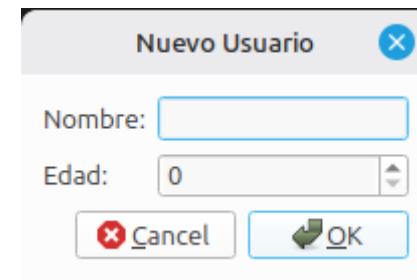
7.7- Diálogos personalizados con QDialogButtonBox

- ▶ Cuando necesitamos un diálogo con campos de datos (formularios), podemos heredar de QDialog y construirlo a medida.
- ▶ Para gestionar los botones de aceptar/cancelar se utiliza la clase QDialogButtonBox, que proporciona una interfaz estandarizada.

```
# Botones de aceptar y cancelar
botones = QDialogButtonBox(QDialogButtonBox.Ok | QDialogButtonBox.Cancel)
botones.accepted.connect(self.validar_y_aceptar)
botones.rejected.connect(self.reject)
```

```
# metodo para validar datos y aceptar
def validar_y_aceptar(self):
    if not self.nom.text().strip():
        QMessageBox.warning(self, "Validación", "El nombre no puede estar vacío.")
        return
    if self.edad.value() < 18:
        QMessageBox.warning(self, "Validación", "Edad debe ser mayor a 18 años.")
        return

    self.accept() # Datos validados y cerramos ventana con éxito
```



“

**8.- Ventana principal, menús,
barra de herramientas y docks
flotantes.**

”

CREAMOS ACCESO RÁPIDO A HERRAMIENTAS

Creación de menús, barras y docs

8.1- Menús, acciones y estado

- ▶ Un menú es un componente estándar que se puede configurar en las ventanas principales QMainWindow.
- ▶ Se ubican en la parte superior de la ventana y permiten a los usuarios acceder a las funcionalidades de las aplicaciones.
- ▶ Hay menús estandarizados como los de Fichero, Edición y Ayuda, cada uno con sus propias jerarquías y árboles de funciones.
- ▶ También ofrecen accesos directos y otras opciones de accesibilidad.

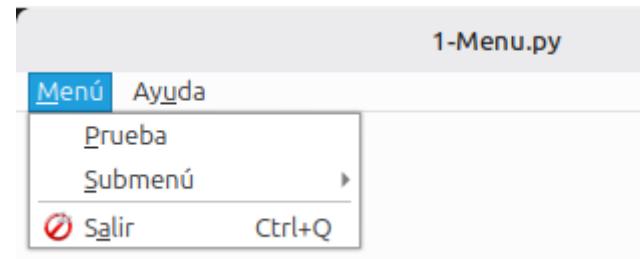
8.1- Menús, acciones y estado

- ▶ Las opciones del menú se llaman acciones porque en realidad son objetos de la clase QAction que estamos creando implícitamente.
- ▶ Vamos a completar la acción de salir con un ícono, un accesor y para que se llame el método close() de la ventana principal:

```
# construimos nuestro menú
self.construir_menu()

def construir_menu(self):
    # Recuperamos la barra de menú
    menu = self.menuBar()

    # Añadimos un menú de archivo
    menu_archivo = menu.addMenu("&Menú")
    # Añadimos una acción de prueba
    menu_archivo.addAction("&Prueba")
    # Añadimos un submenú
    submenu_archivo = menu_archivo.addMenu("&Submenú")
    # Añadimos una acción de prueba
    submenu_archivo.addAction("Subopción &1")
    submenu_archivo.addAction("Subopción &2")
    # Añadimos un separador
    menu_archivo.addSeparator()
    # Añadimos una acción completa
    menu_archivo.addAction(
        QIcon(absPath("exit.png")), "S&alir", self.close, "Ctrl+Q")
```



8.1- Menús, acciones y estado

- ▶ Ahora bien, con el objetivo de reutilizar código es aconsejable crear nuestras propias acciones y luego añadirlas a los menús en lugar de hacerlo implícitamente.
- ▶ Las acciones también permiten configurar lo que se conoce como statusTip para mostrar la utilidad de la acción. Es texto que se muestra en la barra de estado de la ventana principal, una barra que hay que activar.
- ▶ Ahora al pasar el ratón por encima de la acción aparecerá en la parte inferior el texto explicativo.

8.1- Menús, acciones y estado

- Veamos algunos pasos:

```
# Añadimos un menú de archivo
menu_archivo = menu.addMenu("&Menú"
# Añadimos una acción de prueba
menu_archivo.addAction("&Prueba")
```

Creamos un separador (algo visual) y creamos otra opción de menú con un ícono, y la acción, en este caso: Close

Creamos una barra menú, con algunos elementos, indicando la letra a subrayar para posibles atajos

```
# Añadimos un separador
menu_archivo.addSeparator()
# Añadimos una acción completa
menu_archivo.addAction(
    QIcon(absPath("./imagenes/exit.png")), "S&alir", self.close, "Ctrl+Q")
```

```
# También podemos especificar un accesario
accion_info.setShortcut("Ctrl+I")
# Le configuramos una señal para ejecutar un método
accion_info.triggered.connect(self.mostrar_info)
# Añadimos un texto de ayuda
accion_info.setStatusTip("Muestra información irrelevante")
# Añadimos la acción al menú
menu_ayuda.addAction(accion_info)
```

Al pasar por encima de la opción “información”, nos muestra un texto explicativo en la barra de estado. Además de añadir una acción

8.2- Barra de herramientas

- ▶ Las barras de herramientas son componentes estándar para ejecutar funcionalidades de los programas. A diferencia de los menús estas barras son más flexibles y generalmente presentan las opciones de forma visual mediante iconos:



8.2- Barra de herramientas

► Veamos algunos pasos:

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(480, 320)

        self.construir_menu()
        # construimos las herramientas
        self.construir_herramientas()
```

En nuestra ventana principal creamos un menú y una barra de herramientas

En el menú incluimos las acciones deseadas en la ventana principal con su ejecución.

```
def construir_herramientas(self):
    # Creamos una barra de herramientas
    herramientas = QToolBar("Barra de herramientas principal")
    # Podemos agregar la acción salir implícitamente
    herramientas.addAction(
        QIcon(absPath("./imagenes/exit.png")), "S&alir", self.close)
    # O añadir una acción ya creada para reutilizar código
    herramientas.addAction(self.accion_info)
    # La añadimos a la ventana principal
    self.addToolBar(herramientas)
```

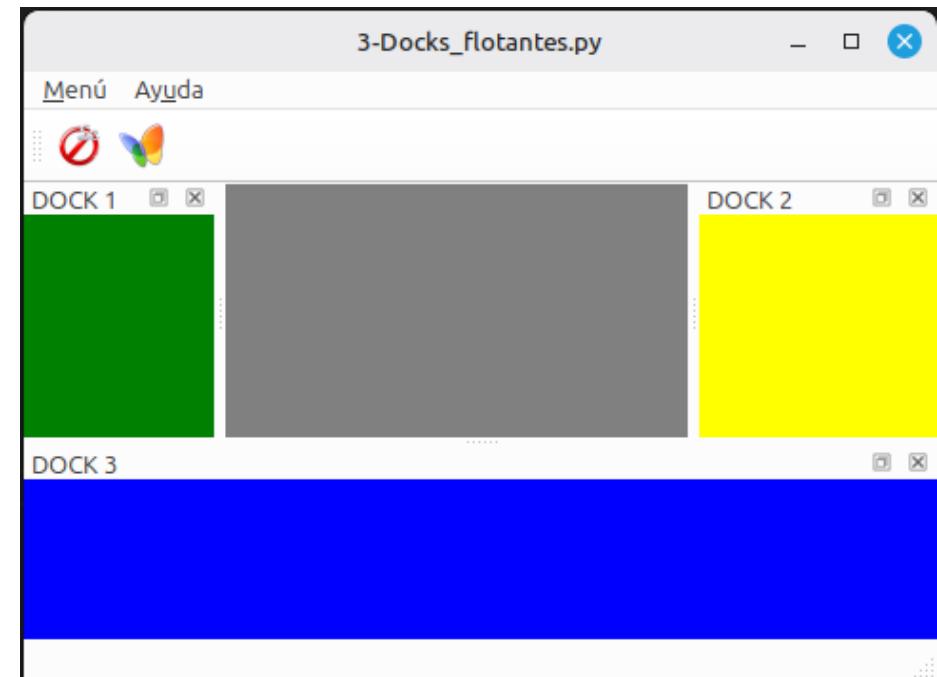
```
# Creamos una acción específica para mostrar información
accion_info = QAction("&Información", self)
# Podemos configurar un ícono en la acción
accion_info.setIcon(QIcon(absPath("./imagenes/info.png")))
# También podemos especificar un accesario
accion_info.setShortcut("Ctrl+I")
# Le configuraremos una señal para ejecutar un método
accion_info.triggered.connect(self.mostrar_info)
# Añadimos un texto de ayuda
accion_info.setStatusTip("Muestra información irrelevante")
# Añadimos la acción al menú
menu_ayuda.addAction(accion_info)

# accesores de clase
self.accion_info = accion_info
```

En la barra de herramientas, creamos una acción para “Salir”, y reutilizamos la acción del menú “Información”.

8.3- Docks flotantes

- ▶ Los docks son contenedores flotantes que se pueden posicionar a los lados de la ventana, desacoplarlos e incluso cerrarlos.
- ▶ Al igual que la ventana principal tiene su método para establecer un widget principal.
- ▶ Los docks tienen un método setWidget para configurar el widget que contendrán.



8.3- Docks flotantes

- Veamos algunos pasos:

```
class MainWindow(QMainWindow):  
    def __init__(self):  
        super().__init__()  
        self.resize(480, 320)  
  
        self.construir_menu()  
        # construimos las herramientas  
        self.construir_herramientas()  
        # añadimos los docks  
        self.construir_docks()  
        # creamos una caja como widget central de la ventana principal  
        self.setCentralWidget(Contenedor("gray"))
```

Esta es la creación del dock

En nuestra ventana principal creamos un menú y una barra de herramientas. Ahora definimos un dock

```
def construir_docks(self):  
    # creamos un dock  
    dock1 = QDockWidget()  
    # le damos un título (optional)  
    dock1.setWindowTitle("DOCK 1")  
    # establecemos el widget que contendrá  
    dock1.setWidget(Contenedor("green"))  
    # ancho mínimo (optional)  
    dock1.setMinimumWidth(100)
```

8.3- Docks flotantes

- ▶ Los docks son super flexibles.
- ▶ Al cerrarlo lo perdemos y deberíamos proveer de alguna forma de crearlo de nuevo, o también podemos limitar sus características:

```
# el dock1 no se puede ni mover, ni cerrar, ni hacer flotar  
dock1.setFeatures(QDockWidget.NoDockWidgetFeatures)
```

Constante	Valor	Descripción
QDockWidget::DockWidgetClosable	0x01	El widget del dock se puede cerrar.
QDockWidget::DockWidgetMovable	0x02	El widget del dock puede ser movido entre diferentes docks por el usuario.
QDockWidget::DockWidgetFloatable	0x04	El widget acoplable se puede separar de la ventana principal y flotar como una ventana independiente.
QDockWidget::DockWidgetVerticalTitleBar	0x08	El widget del dock muestra una barra de título vertical en su lado izquierdo. Esto se puede utilizar para aumentar la cantidad de espacio vertical en un QMainWindow .
QDockWidget::NoDockWidgetFeatures	0x00	El widget del dock no se puede cerrar, mover ni hacer flotar.

8.3- Docks flotantes

- ▶ También podemos controlar su tamaño:
- ▶ Y añadir más docks en otras posiciones para juguetear con ellos

```
# ancho mínimo (opcional)
dock1.setMinimumWidth(100)

# tamaños (opcionales)
dock1.setMinimumWidth(125)
dock1.setMinimumHeight(100)
dock1.setMinimumSize(125, 100)
```

```
# creamos más docks para jugar con ellos
dock2 = QDockWidget()
dock2.setWindowTitle("DOCK 2")
dock2.setWidget(Contenedor("yellow"))
dock2.setMinimumSize(125, 100)
self.addDockWidget(Qt.RightDockWidgetArea, dock2)

dock3 = QDockWidget()
dock3.setWindowTitle("DOCK 3")
dock3.setWidget(Contenedor("blue"))
dock3.setMinimumSize(125, 100)
self.addDockWidget(Qt.BottomDockWidgetArea, dock3)
```

“

9.- Creación y control de subventanas.

”

CREAMOS Y GESTIONAMOS VENTANAS SECUNDARIAS

Controlar el uso de ventanas secundarias

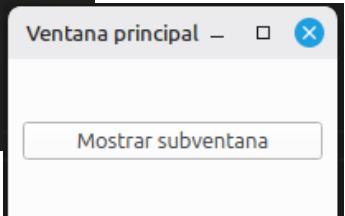
9.1- Creación de subventanas

- ▶ Para crear una subventana partimos de una clase heredada de QWidget que instanciaremos en el método mostrar_subventana.
- ▶ En caso de no instanciar correctamente, la ventana se abre y se cierra de forma automática. No nos da tiempo a saber si se ha creado

```
# botón para abrir la subventana
boton_mostrar = QPushButton("Mostrar subventana")
boton_mostrar.clicked.connect(self.mostrar_subventana)
layout.addWidget(boton_mostrar)

# Muestra la ventana, pero no la implementa completamente,
# así que la ventana desaparece rápidamente
def mostrar_subventana(self):
    print("Subventana abierta")

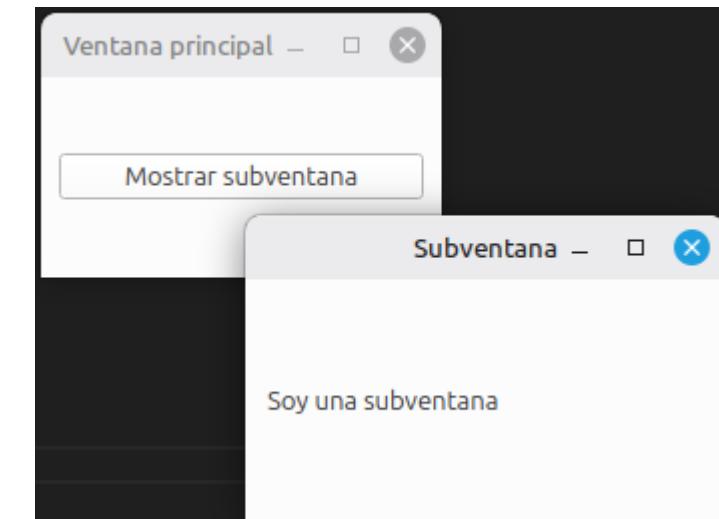
class Subventana(QWidget):
    def __init__(self):
        super().__init__()
        # Le damos un tamaño y un título
        self.resize(240, 120)
        self.setWindowTitle("Subventana")
        # creamos una etiqueta
        etiqueta = QLabel("Soy una subventana")
        # creamos un layout y añadimos la etiqueta
        layout = QVBoxLayout()
        layout.addWidget(etiqueta)
        # asignamos el layout al widget
        self.setLayout(layout)
```



9.1- Creación de subventanas

- ▶ Para poder ver la ventana secundaria, sustituimos el método mostrar_subventana, de forma que creemos una ventana que forme parte de la clase, como atributo de la ventana principal.

```
# Muestra la ventana, pero no la implementa completamente,  
# así que la ventana desaparece rápidamente  
def mostrar_subventana(self):  
    # print("Subventana abierta")  
  
    # Ahora sí implementamos la subventana correctamente  
    # ya que la guardamos como atributo de la clase  
    def mostrar_subventana(self):  
        self.subventana = Subventana()  
        self.subventana.show()
```



9.2- Subventana persistente

- ▶ Es importante dejar claro que cada vez que presionamos el botón que crea la subventana, se crea una nueva que sobrescribe a la anterior.
- ▶ No es una mala opción, pero si queremos interactuar con la subventana y que los cambios se almacenen en ella, entonces no podemos tomarnos el lujo de sobrescribirla.
- ▶ Para evitar sobrescribirla, crearemos la instancia solamente si la variable de clase es nula, con esto nos aseguramos de tener almacenada la primera instancia de la subventana para no perder su contenido.

9.2- Subventana persistente

- Veamos el cambio en el código:

```

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

        # iniciamos la variable de la subventana nula por defecto
        self.subventana = None

    def mostrar_subventana(self):
        # si no hay ninguna instancia la guardamos
        if not self.subventana:
            self.subventana = Subventana()

        # y la mostramos
        self.subventana.show()

class Subventana(QWidget):
    def __init__(self):
        super().__init__()
        self.resize(240, 120)
        self.setWindowTitle("Subventana")

        # creamos una etiqueta con texto aleatorio que demuestra
        # la persistencia de la subventana
        etiqueta = QLabel(f"Soy una subventana... {random.randint(0, 100)}")

        layout = QVBoxLayout()
        layout.addWidget(etiqueta)
        self.setLayout(layout)

```

- Al tenerla guardada podemos interactuar con ella, por ejemplo, para esconderla desde la ventana principal:

```

# botón para cerrar la subventana
boton_cerrar = QPushButton("Ocultar subventana")
boton_cerrar.clicked.connect(self.ocultar_subventana)
layout.addWidget(boton_cerrar)

def ocultar_subventana(self):
    # si la subventana existe y es visible la escondemos
    if self.subventana and self.subventana.isVisible():
        self.subventana.hide()

```

9.2- Subventana persistente

- ▶ Pero bien pensado, dado que solo necesitamos una instancia de la subventana, podríamos crearla desde el principio en la ventana principal.

```
# creamos una instancia de la subventana al principio  
self.subventana = Subventana()
```

- ▶ Al tener la subventana desde el principio podemos acceder a sus métodos show y hide, pero deberemos llamarlos como funciones anónimas.
- ▶ Las funciones anónimas se crean en tiempo de ejecución y se asegura que la subventana existe antes de llamar a sus métodos.

9.2- Subventana persistente

- ▶ El código podría ser:

```
# creamos una instancia de la subventana al principio
self.subventana = Subventana()

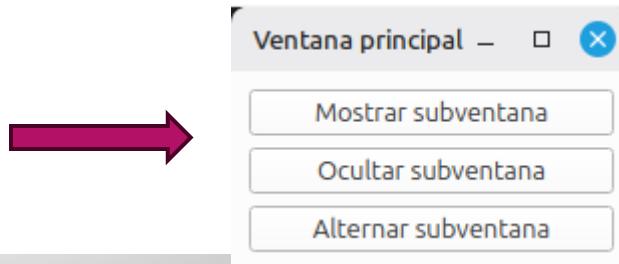
##### ===> Borramos los métodos mostrar_ventana() y ocultar_ventana()

# botón para mostrar la subventana
boton_mostrar = QPushButton("Mostrar subventana")
boton_mostrar.clicked.connect(self.subventana.show)
layout.addWidget(boton_mostrar)

# botón para ocultar la subventana
boton_ocultar = QPushButton("Ocultar subventana")
boton_ocultar.clicked.connect(self.subventana.hide)
layout.addWidget(boton_ocultar)
```

- ▶ Vamos a crear un tercer botón alternador en única línea que pueda mostrar y ocultar la subventana aprovechando el potencial de estas funciones:

```
# botón para alternar la subventana
boton_alternador = QPushButton("Alternar subventana")
boton_alternador.clicked.connect(
    lambda: self.subventana.hide()
    if self.subventana.isVisible()
    else self.subventana.show())
layout.addWidget(boton_alternador)
```



“

10.- Tematización. Estilos, paletas, íconos, ...

”

CREAMOS APLICACIONES CON UN ESTILO ADECUADO

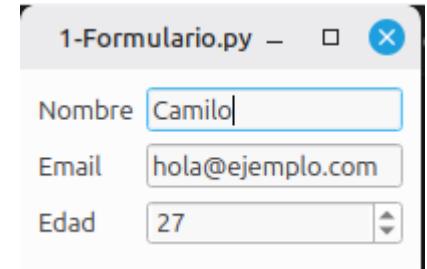
Controlar el estilo de nuestra aplicación

10.1- Estilos

- ▶ Los estilos modifican la estética de los componentes. Por defecto Qt aplica estilos específicos para cada plataforma para integrar la aplicación visualmente. Esa es la razón por la que el mismo programa se verá diferente en Windows, Linux y Mac.
- ▶ Los estilos se pueden personalizar para no hacerlos dependientes de la plataforma y de hecho el propio Qt tiene un tema llamado Fusion que provee una estética multiplataforma y moderna.
- ▶ Veamos un formulario con todo tipo de widgets donde podamos apreciar los cambios visuales al cambiar los estilos:

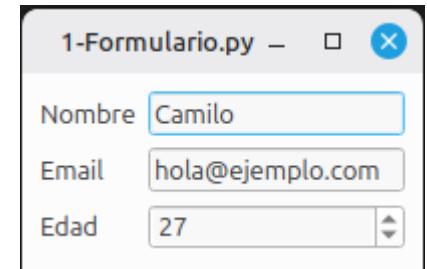
10.1- Estilos

- ▶ Se ha creado un formulario base, sin ningún tipo de estilo.
- ▶ Se ha repetido la operación, pero activando el estilo “Fusion”.
- ▶ Nos encontramos dos ventanas idénticas, pues en Linux, este estilo proporciona una apariencia de escritorio uniforme e igual a la plataforma. En Windows hay pequeños detalles del cambio de estilo.



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

    # estilo fusion
    app.setStyle("Fusion")
```



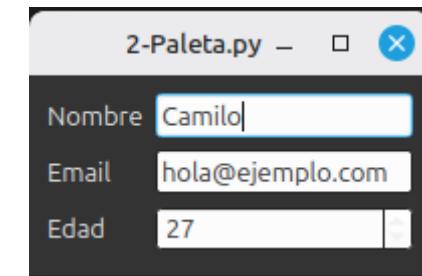
10.2- Paletas

- ▶ La selección de colores que utiliza Qt para dibujar los componentes se maneja en paletas.
- ▶ Se trata de un diálogo específico, vista anteriormente.
- ▶ Las paletas tienen accesores para establecer los colores de los diferentes componentes. Les damos unos valores predeterminados y vemos el resultado.

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

    # creamos nuestra paleta de colores
    paleta = QPalette()
    paleta.setColor(QPalette.Window, QColor(51, 51, 51))
    paleta.setColor(QPalette.WindowText, QColor(235, 235, 235))

    # activamos la paleta en la aplicación
    app.setPalette(paleta)
```



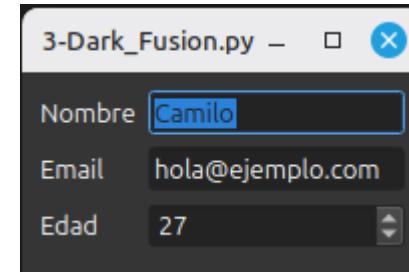
10.2- Dark Fusion

- ▶ Probemos una paleta llamada Dark Fusion, bastante atractiva para cambiar la apariencia a modo oscuro.

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

    # dark fusion https://gist.github.com/lschmierer/443b8e21ad93e2a2d7eb
    app.setStyle("Fusion")
    dark_fusion = QPalette()
    dark_fusion.setColor(QPalette.Window, QColor(53, 53, 53))
    dark_fusion.setColor(QPalette.WindowText, Qt.white)
    dark_fusion.setColor(QPalette.Base, QColor(35, 35, 35))
    dark_fusion.setColor(QPalette.AlternateBase, QColor(53, 53, 53))
    dark_fusion.setColor(QPalette.ToolTipBase, QColor(25, 25, 25))
    dark_fusion.setColor(QPalette.ToolTipText, Qt.white)
    dark_fusion.setColor(QPalette.Text, Qt.white)
    dark_fusion.setColor(QPalette.Button, QColor(53, 53, 53))
    dark_fusion.setColor(QPalette.ButtonText, Qt.white)
    dark_fusion.setColor(QPalette.BrightText, Qt.red)
    dark_fusion.setColor(QPalette.Link, QColor(42, 130, 218))
    dark_fusion.setColor(QPalette.Highlight, QColor(42, 130, 218))
    dark_fusion.setColor(QPalette.HighlightedText, QColor(35, 35, 35))
    dark_fusion.setColor(QPalette.Active, QPalette.Button, QColor(53, 53, 53))
    dark_fusion.setColor(QPalette.Disabled, QPalette.ButtonText, Qt.darkGray)
    dark_fusion.setColor(QPalette.Disabled, QPalette.WindowText, Qt.darkGray)
    dark_fusion.setColor(QPalette.Disabled, QPalette.Text, Qt.darkGray)
    dark_fusion.setColor(QPalette.Disabled, QPalette.Light, QColor(53, 53, 53))
    # activamos la paleta en la aplicación
    app.setPalette(dark_fusion)

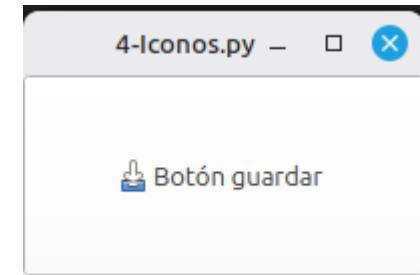
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```



10.3- Iconos

- ▶ Ya hemos utilizado algunos iconos cargándolos como recursos externos, pero Qt incluye un set de iconos predeterminados.
- ▶ Podemos hacer uso de ellos de la siguiente forma.

```
class MainWindow(QMainWindow):  
    def __init__(self):  
        super().__init__()  
  
        # recuperamos el icono de la librería estandard de la ventana  
        icono = self.style().standardIcon(QStyle.SP_DialogSaveButton)  
        # lo podemos asignar a un botón  
        boton = QPushButton(icono, "Botón guardar")  
  
        self.setCentralWidget(boton)
```



10.3- Iconos

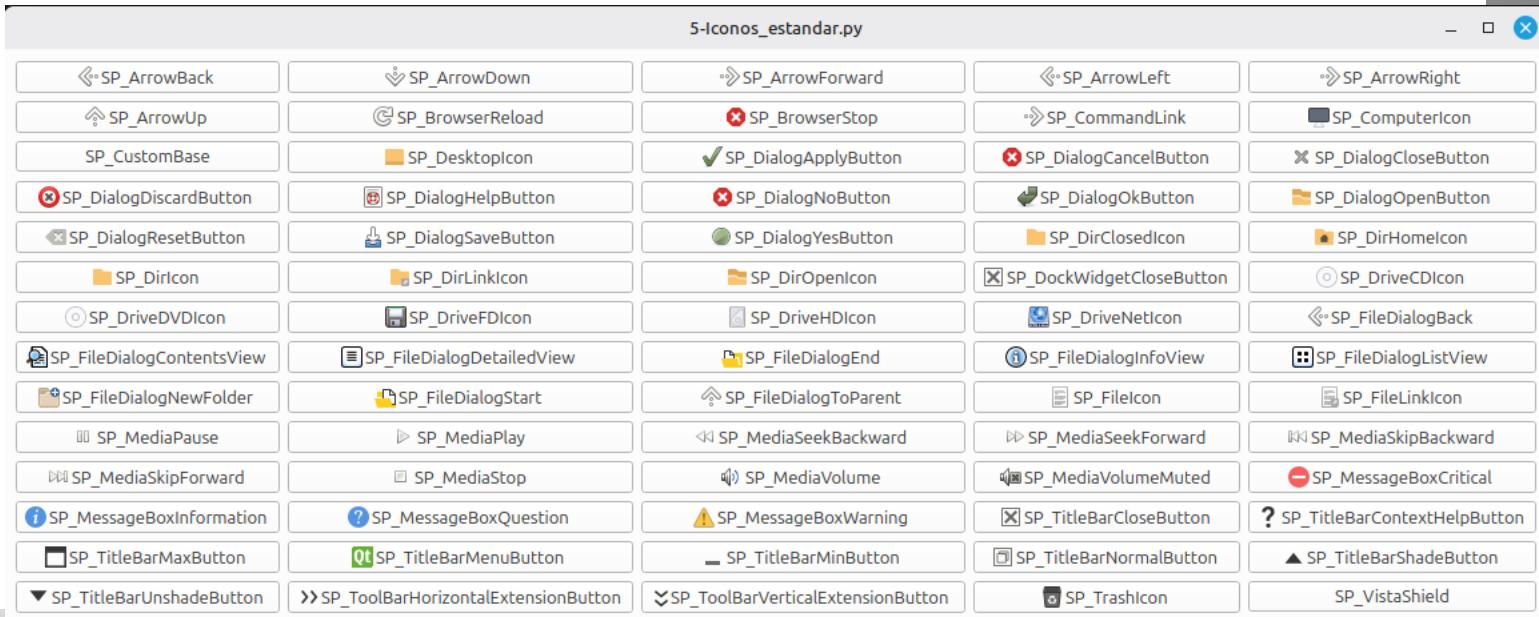
- Veamos un programa para visualizar dinámicamente los iconos de la librería estándar.

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

        # creo un layout en cuadrícula
        layout = QGridLayout()

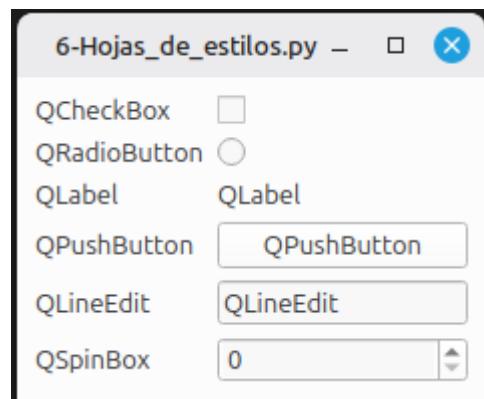
        # recorro los iconos con un contador de posición
        for contador, nombre in enumerate(iconos):
            # recupero el ícono a partir de su nombre
            ícono = self.style().standardIcon(getattr(QStyle, nombre))
            # creo un botón con el ícono y su nombre del ícono
            botón = QPushButton(ícono, nombre)
            # añado el botón en una cuadrícula de 5 columnas
            # divido el contador entre 5 para conseguir la fila
            # con el módulo de la división entre 5 conseguiré la columna
            layout.addWidget(botón, contador // 5, contador % 5)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)
```



10.4- Qt Style Sheets

- ▶ Veamos, por último, las hojas de estilo de Qt, o abreviado QSS.
- ▶ QSS es una forma de añadir estilo a los widgets utilizando prácticamente la misma sintaxis que CSS.

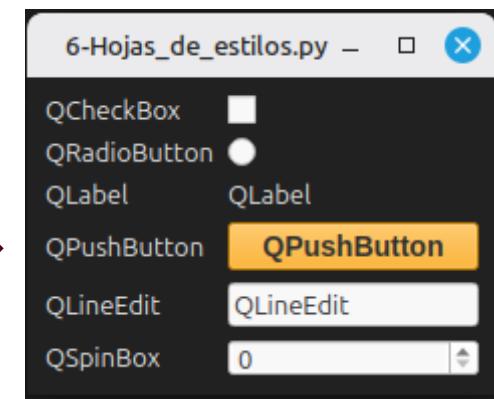


Antes de incluir estilos

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

        # estilos QSS
        self.setStyleSheet("""
            QMainWindow {background-color: #212121; }
            QLabel {color: #e9e9e9; }
            QPushButton {background-color: orange;
                        font-family: "Arial";
                        font-size: 14px;
                        font-weight: bold; }
            """)


```

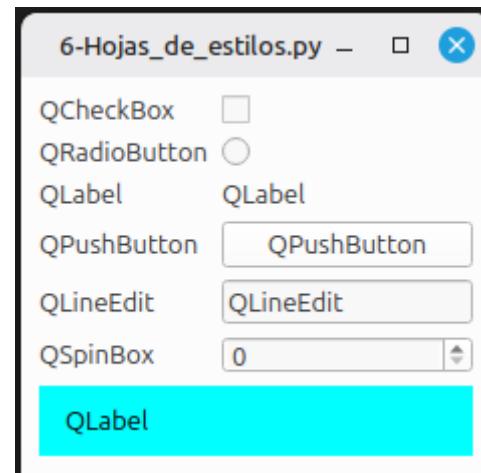


Después de incluir estilos

10.4- Qt Style Sheets

- ▶ Ahora bien, estos estilos son globales y afectan a todas las instancias. Si queremos estilizar una sola instancia podemos otorgarle un identificador
- ▶ Y referirnos a ella en QSS usando la almohadilla igual que en CSS:

```
self.setStyleSheet("""  
    #etiqueta {background-color: cyan;  
                padding: 10px;  
                color: black; }  
"""")  
  
etiqueta = QLabel("QLabel")  
etiqueta.setObjectName("etiqueta")  
formulario.addRow(etiqueta)
```



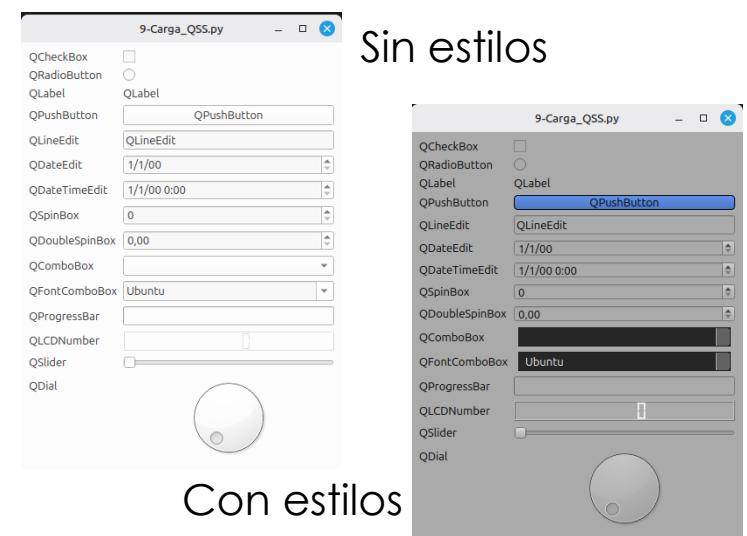
10.5- Cargando ficheros QSS

- ▶ Veamos como cargar ficheros QSS para no tener que escribir el código en el propio programa.
- ▶ Para cargar los estilos me ayudo de la función absPath, abro los ficheros de estilo como si fueran texto, leo su contenido y lo vuelco al método setStyleSheet de la ventana:

```
def cargarQSS(self, file):
    # guardamos la ruta absoluta al fichero
    path = absPath(file)
    # intentamos abrirlo y volcar el contenido
    try:
        with open(path) as styles:
            self.setStyleSheet(styles.read())
        # si hay algún fallo lo capturamos con una excepción genérica
    except:
        print("Error abriendo estilos", path)
```

```
# cargamos los estilos del fichero
self.cargarQSS("qss/Ubuntu.qss")
self.cargarQSS("qss/ElegantDark.qss")
self.cargarQSS("qss/ChatBee.qss")
self.cargarQSS("qss/EasyCode.qss")
```

Función para cargar ficheros QSS



10.4- Qt Style Sheets

- ▶ Ahora bien, estos estilos son globales y afectan a todas las instancias. Si queremos estilizar una sola instancia podemos otorgarle un identificador
- ▶ Y referirnos a ella en QSS usando la almohadilla igual que en CSS:

```
self.setStyleSheet("""  
    #etiqueta {background-color: cyan;  
                padding: 10px;  
                color: black; }  
"""")  
  
etiqueta = QLabel("QLabel")  
etiqueta.setObjectName("etiqueta")  
formulario.addRow(etiqueta)
```



“

11.- Diseñar con Qt Designer.
Crear diseños con ayuda una
interfaz grafica.

”

CREAMOS INTERFACES DE USUARIO CON AYUDA DE QT DESIGNER

Conocer Qt Designer

11.1- Aplicación Qt Designer

- ▶ Para diseñar una interfaz utilizando esta aplicación, debemos tener cuidad de la plataforma que utilice el usuario, porque existen cambios importantes según donde se ejecuta.
- ▶ Como estamos trabajando con Linux, para ejecutar la aplicación, debemos acceder a nuestro entorno virtual, y seguir la ruta:

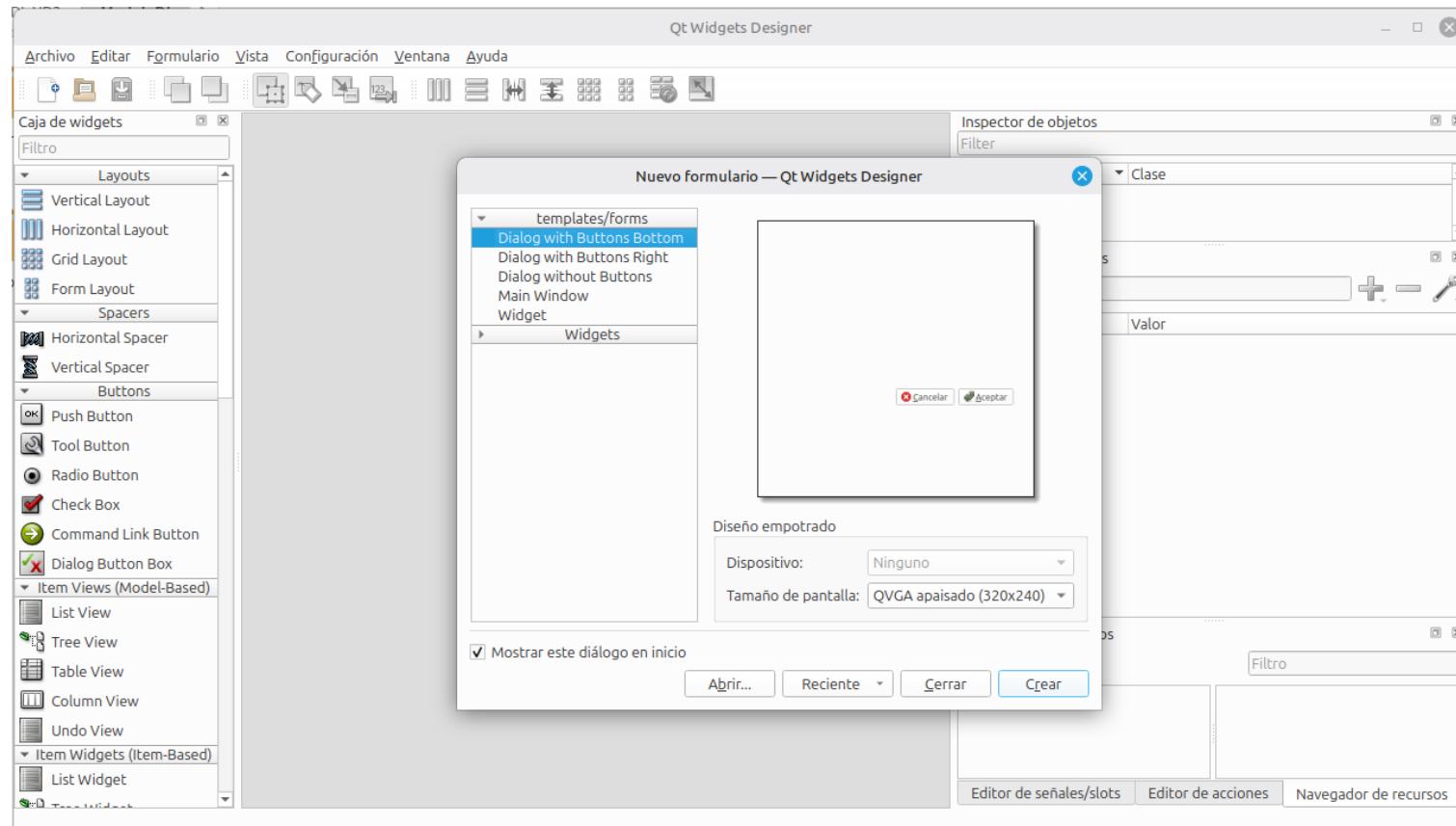
/lib/python3.12/site-packages/PySide6\$

Nuestro entorno virtual se creo en una carpeta y utilizamos el nombre de ModuloDI (Ova), así que la ruta es: **~/DI_UD2/ModuloDI/lib/python3.12/site-packages/PySide6\$**

- ▶ Podemos ejecutar ahora, la instrucción designer que nos abrirá una aplicación para diseñar interfaces. **./designer**
- ▶ Realizaremos nuestra primera aplicación con una actividad guiada.

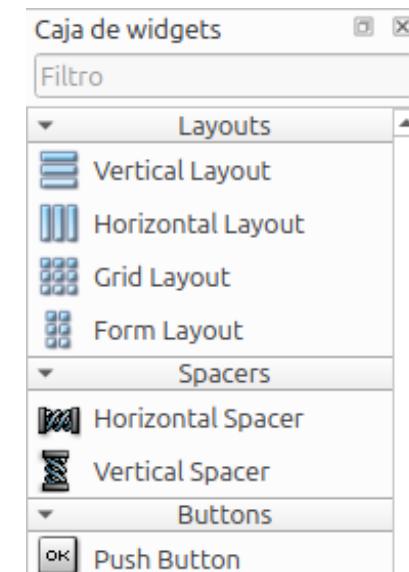
11.1- Aplicación Qt Designer

- ▶ Una vez ejecutada la instrucción, nos aparece la siguiente aplicación:



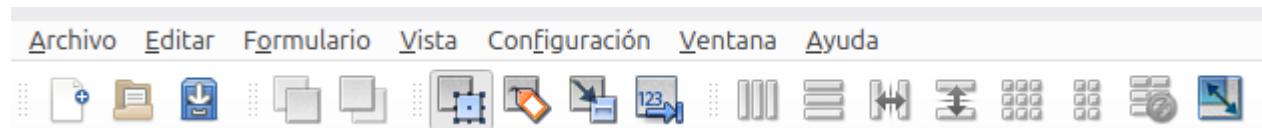
11.1- Aplicación Qt Designer

- ▶ Vamos a elegir Main Windws y nos aparece en el centro la vista de diseño con la ventana principal, con barra de menú y barra de estado por defecto.
- ▶ Veamos que otros elementos nos encontramos:
 - A la izquierda tenemos la caja de Widgets, con muchos elementos que podemos utilizar.

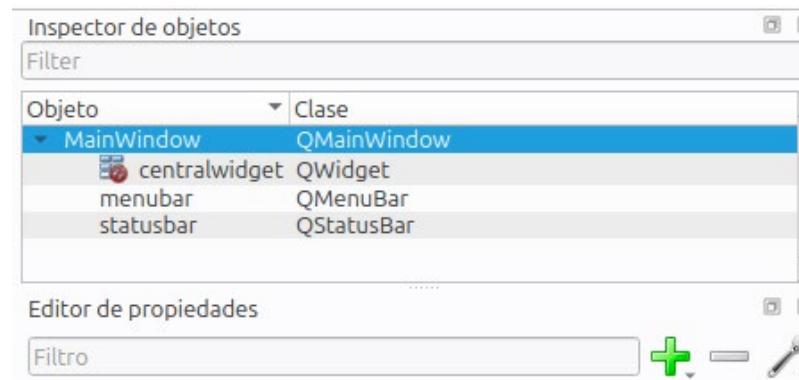


11.1- Aplicación Qt Designer

- Arriba el control de elementos y disposiciones.

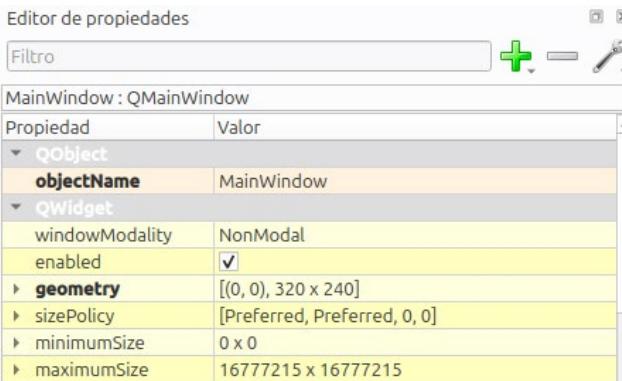


- A la derecha encontramos de arriba a abajo:
 - El inspector de objetos en una vista jerárquica.

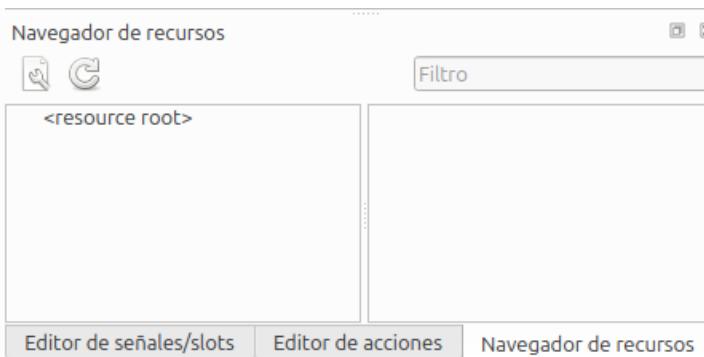


11.1- Aplicación Qt Designer

- El editor de propiedades del componente seleccionado.

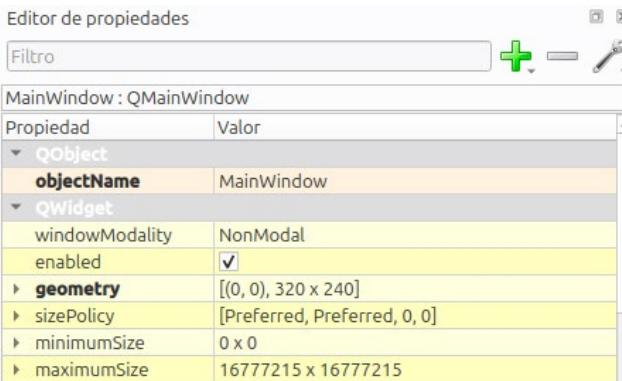


- Y el navegador de recursos del programa.

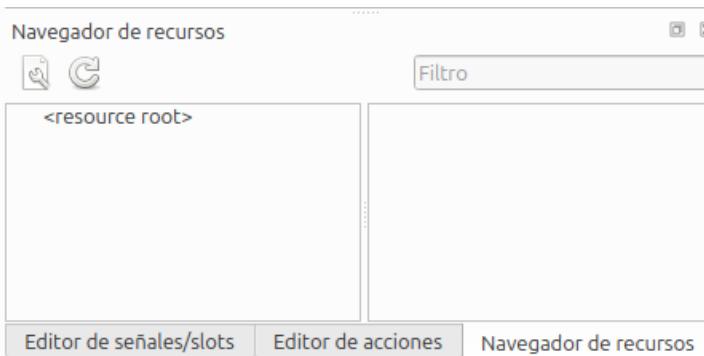


11.1- Primer diseño en Qt Designer

- El editor de propiedades del componente seleccionado.



- Y el navegador de recursos del programa.



11.2- Crear ejecutable

- ▶ Para finalizar con esta unidad, vamos a convertir nuestra aplicación en un fichero ejecutable que contenga todas las dependencias instaladas en su entorno virtual de Python.
- ▶ Para ello vamos a utilizar uno de los paquetes instalados en la primera práctica guiada realizada en esta unidad. Se trata del paquete auto-py-to-exe, aunque si intentamos ejecutar nos indica que falta instalar una dependencia: python3-tk.

La ruta es: **~/DI_UD2/ModuloDI/bin\$./auto-py-to-exe**

- ▶ Debemos instalarla:

```
alu@Pc-IAW:~/DI_UD2/ModuloDI/bin$ sudo apt-get install python3-tk
```

11.2- Crear ejecutable

- Vamos la ejecución completa:

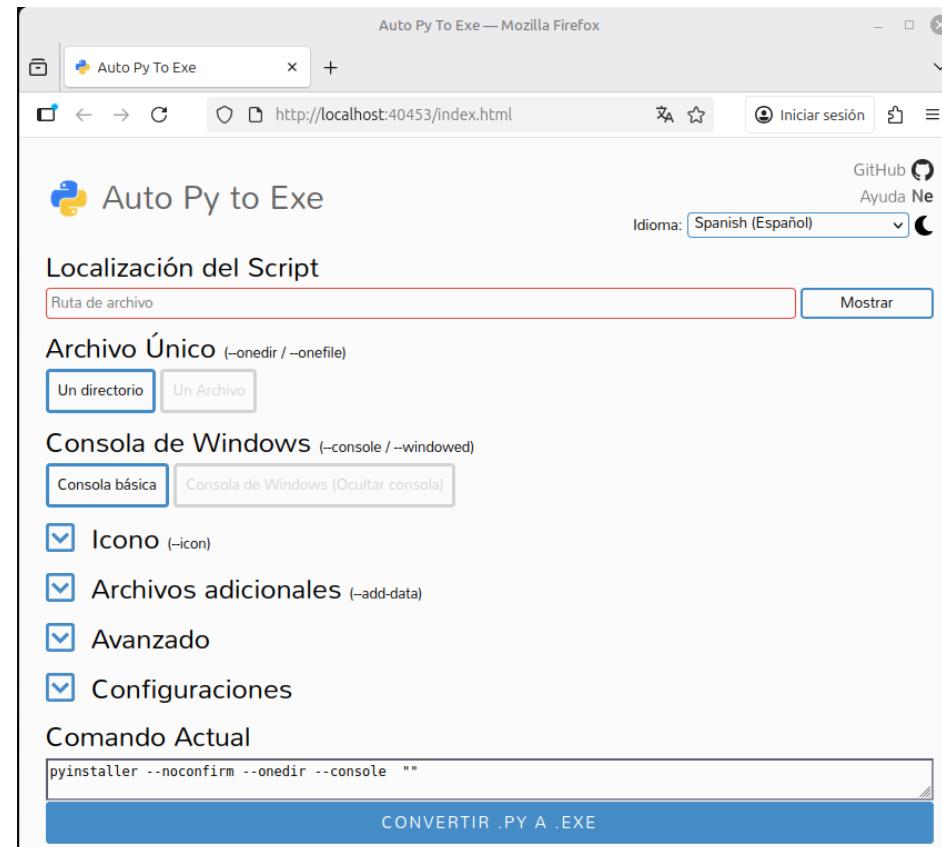
```
alu@Pc-IAW:~/DI_UD2/ModuloDI/bin$ ls
activate          pyi-bindepend      pyside6-project
activate.csh      pyi-grab_version  pyside6-qml
activate.fish     pyi-makespec      pyside6-qmlcachegen
Activate.ps1       pyinstaller      pyside6-qmlformat
auto-py-to-exe    pyi-set_version   pyside6-qmlimportscanner
autopytoexe       pyside6-android-deploy  pyside6-qmllint
bottle           pyside6-assistant   pyside6-qmlls
bottle.py         pyside6-balsam     pyside6-qmlyperegistrar
futurize          pyside6-balsamui    pyside6-qsb
normalizer        pyside6-deploy     pyside6-qtpy2cpp
pasteurize        pyside6-designer   pyside6-rcc
pip               pyside6-genpyi     pyside6-svgtoqml
pip3              pyside6-linguist    pyside6-uic
pip3.12           pyside6-lrelease   python
__pycache__        pyside6-lupdate    python3
pvi-archive       pyside6-metaobjectdump python3.12
viewer           pyside6-viewer

alu@Pc-IAW:~/DI_UD2/ModuloDI/bin$ ./auto-py-to-exe
Error: tkinter not found
For linux, you can install tkinter by executing: "sudo apt-get install python3-tk"
alu@Pc-IAW:~/DI_UD2/ModuloDI/bin$ sudo apt-get install python3-tk
[sudo] contraseña para alu:
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias... Hecho
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  blt libtk8.6 tk8.6-blt2.5
Paquetes sugeridos:
  blt-demo tk8.6 tix python3-tk-dbg
Se instalarán los siguientes paquetes NUEVOS:
  blt libtk8.6 python3-tk tk8.6-blt2.5
0 actualizados, 4 nuevos se instalarán, 0 para eliminar y 0 no actualizados.
Se necesita descargar 1.516 kB de archivos.
Se utilizarán 4.929 kB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n]

alu@Pc-IAW:~/DI_UD2/ModuloDI/bin$ ./auto-py-to-exe
The interface is being opened in your default browser
Please do not close this terminal while using auto-py-to-exe - the process will
end when the window is closed
```

11.2- Crear ejecutable

- ▶ Como se puede comprobar, podemos cambiar el idioma y en mi caso, solo debo de indicar que script que quiero convertir en ejecutable.
- ▶ Busco el python que quiero convertir y pincho en el botón de convertir.



11.2- Crear ejecutable

- ▶ Esto puede tardar un poco.

```
330274 INFO: Building EXE from EXE-00.toc
330306 INFO: Copying bootloader EXE to /tmp/tmpb5mui5j/_build/1-Formulario/1-Formulario
330316 INFO: Appending PKG archive to custom ELF section in EXE
330498 INFO: Building EXE from EXE-00.toc completed successfully.
330528 INFO: checking COLLECT
330590 INFO: Building COLLECT because COLLECT-00.toc is non existent
330599 INFO: Building COLLECT COLLECT-00.toc
334320 INFO: Building COLLECT COLLECT-00.toc completed successfully.
334336 INFO: Build complete! The results are available in: /tmp/tmpb5mui5j/_application

Moving project to: /home/alu/DI_UD2/ModuloDI/bin/output
Complete.
```

Algo fue mal con tu exe? Lea [este post](#) contiene posibles problemas comunes y posibles soluciones.

LIMPIAR SALIDA

ABRIR CARPETA DE DESTINO

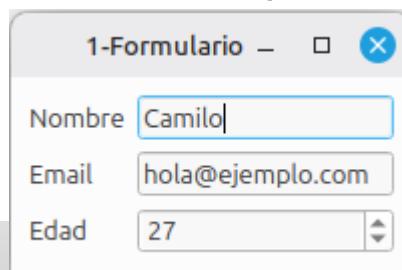
- ▶ Una vez finalizado, podemos abrir la carpeta donde se ha creado nuestro ejecutable

11.2- Crear ejecutable

- ▶ Si nos fijamos un poco, nos ha creado una carpeta en la ruta:
~/DI_UD2/ModuloDI/bin/output\$
- ▶ Tiene el nombre del script que queríamos convertir. En su interior nos aparece el ejecutable y una carpeta con todas las dependencias.

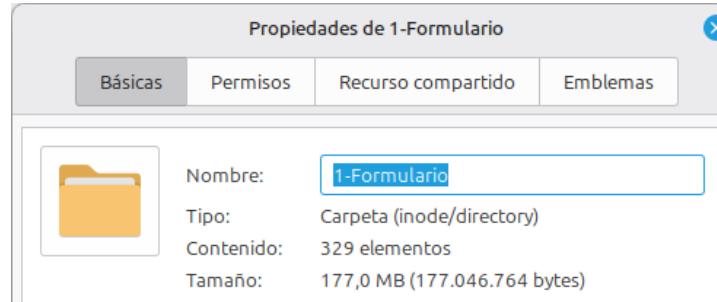


- ▶ Podemos ejecutar esta nueva aplicación a ver que ocurre:



11.2- Crear ejecutable

- ▶ Por último, podemos mover esta carpeta a donde deseemos, pero si nos fijamos, el tamaño puede ser considerable, debido a las dependencias.



- ▶ Se recomienda que para poder moverla, se comprima a gusto del usuario.