Qt, PySide y PyQt

`Qt` es un framework ámpliamente utilizado para desarrollar programas con interfaces gráficas de usuario:

- Está programado en C++, por tanto es muy rápido.
- Es multiplataforma, funciona en diferentes sistema operativos.
- Es orientado a objetos, fácil de empezar a utilizar y aprender.
- Es software libre y código abierto, por tanto su uso es seguro.
- Ofrece licencias públicas, permitiendo su uso de forma gratuita.

`PySide` y `PyQt` son bindings o puentes que permiten utilizar Qt, programado en C++, a través de Python.
PySide es el binding oficial desarrollado por `The Qt Company` con una licencia pública, en cambio PyQt está desarrollado por la firma `Riverbank Computing` y tiene una licencia comercial de $550.
Antiguamente `PySide` estaba por detrás de `PyQt` en características, razón por la cuál se extendió el uso de la segunda alternativa, pero gracias al auge de Python y al apoyo de `The Qt Company`, actualmente la suite de `PySide` contiene todo lo necesario para desarrollar programas gráficos en Python con las ventajas de una licencia pública.

Primera aplicación QApplication

Estructura básica de un programa en PySide usando el componente base de Qt llamado `QWidget`, un widget vacío:

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

# Creamos una aplicación para gestionar la interfaz
app = QApplication(sys.argv)

# Creamos un widget para generar la ventana
window = QWidget()

# Mostramos la ventana, se encuentra oculta por defecto
window.show()

# Iniciamos el bucle del programa
sys.exit(app.exec_())
```

Ejecutamos el programa en la terminal o en Visual Studio Code con la extensión Code Runner `F1 > Run code`:

```
python main.py
```

- `QApplication`: Es el núcleo de un programa en Qt, se requiere para manejar el bucle de la aplicación, encargado de gestionar todas las interacciones con la interfaz gráfica de usuario.

Una aplicación requiere por lo menos un widget para mostrar algo en pantalla. Todos los widgets que heredan de `QWidget` se pueden visualizar como ventanas en sí mismos:

```
import sys

from PySide6.QtWidgets import QApplication, QPushButton

app = QApplication(sys.argv)

# Esta vez la ventana la maneja un widget de tipo botón
window = QPushButton("Hola mundo")
window.show()

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

Como podéis observar esto no es muy útil, ya que toda la ventana es el botón 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.

```
import sys

from PySide6.QtWidgets import QApplication, QMainWindow

app = QApplication(sys.argv)

# Ahora la ventana la gestiona el widget de ventana principal
window = QMainWindow()

# Damos un título a la ventana principal
window.setWindowTitle("Hola mundo")

window.show()

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

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

```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
import sys

app = QApplication(sys.argv)
window = QMainWindow()
window.setWindowTitle("Hola mundo")

# Guardamos el botón en una variable
button = QPushButton("Soy un botón")
# Establecemos el botón como widget central de la ventana principal
```

```
window.setCentralWidget(button)

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

Superclases y subclases

En la próxima lección vamos a transformar el código de nuestro programa a clases y objetos. Para hacerlo necesitaremos trabajar con la herencia de clases, así que 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()
```

La subclase puede sobreescribe 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):
        print(f"Soy Hijo")


hijo = Hijo()
```

Ahora bien, en algunas ocasiones quizá nos interesa no sobreescribir completamente, sino extender una funcionalidad de la superclase. Cuando necesitemos este comportamiento podemos hacer uso de la función `super()`, un accesor directo a la superclase:

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

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


hijo = Hijo()
```

Queda claro entonces que también es posible extender el comportamiento de una superclase sin sobreescribirlo si hacemos uso del accesor `super()`.

¿Y qué pasa con la herencia múltiple? ¿Cómo se comportaría `super()` si heredamos de más de una superclase?

```
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")


hijo = Hijo()
```

Pues en este caso se sigue la lógica de la prioridad de herencia, teniendo más prioridad la clase de izquierda, y por lo tanto `super()` es el accesor de `Madre`.
Si quisiéramos extender el comportamiento del padre, en lugar de `super()` utilizaremos su propio nombre:

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

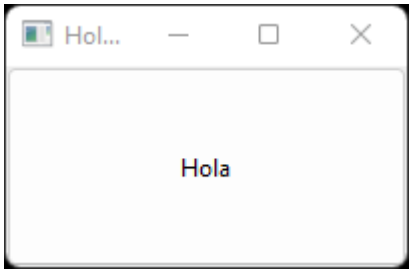
Sin embargo aquí deberemos pasar `self` al método porque `super` lo hace implícitamente.

Para rizar el rizo 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")
```

Con esto queda repasado el concepto de herencia múltiple, necesario para extender los componentes de Pyside.

Primera aplicación usando POO

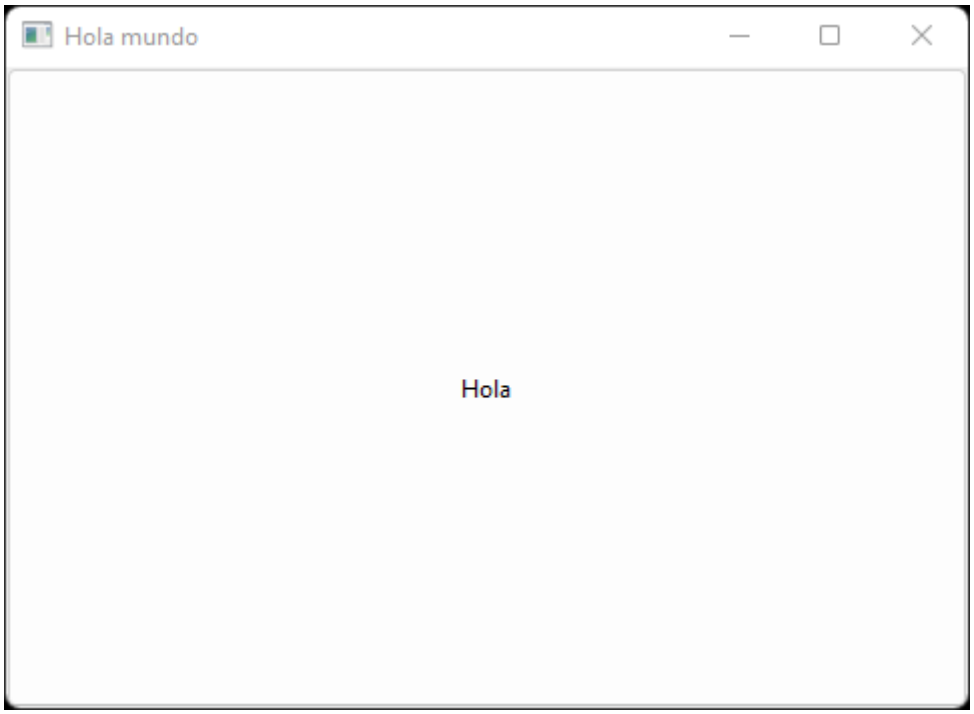



```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
import sys
```

```
class MainWindow(QMainWindow):

    """
    Creamos nuestra propia clase MainWindow heredando de QMainWindow
    """

    # Creamos la ventana en el constructor a partir de una QMainWindow
    def __init__(self):

        # Con super ejecutamos su propio constructor
        # Así se crea la ventana en su propia instancia self
        super().__init__()

        # Damos un título al programa
        self.setWindowTitle("Hola mundo")

        # Guardamos el botón en una variable
        button = QPushButton("Hola")

        # Establecemos el botón como widget central de la ventana
principal
        self.setCentralWidget(button)


# Si ejecutamos el propio script como programa principal
if __name__ == "__main__":
    # Creamos la aplicación
    app = QApplication(sys.argv)
    # Creamos nuestra ventana principal
    window = MainWindow()
    # Mostramos la ventana
    window.show()
    # Iniciamos el bucle del programa
    sys.exit(app.exec_())
```

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`.

Tamaño de los widgets QSize



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:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
from PySide6.QtCore import QSize # Nuevo
import sys

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Hola mundo")
        button = QPushButton("Hola")
        self.setCentralWidget(button)

        # Tamaño mínimo del widget
        self.setMinimumSize(QSize(480, 320))
        # Tamaño máximo del widget
        self.setMaximumSize(QSize(480, 320))
        # Tamaño fijo del widget
        self.setFixedSize(QSize(480, 320))


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

Alternativamente, si no queremos complicarnos la vida podemos utilizar el método `resize` de la ventana:

```
from PySide6.QtWidgets import QApplication, QMainWindow, QPushButton
import sys

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Hola mundo")
```

```
        button = QPushButton("Hola")
        self.setCentralWidget(button)

        # Redimensión simple
        self.resize(480, 320)


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

El inconveniente es que este método no da tanto juego como establecer las reglas independientes.

Señales y receptores (Signals y Slots)

Tenemos un botón en nuestro programa pero no hace nada.

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.

Pues bien, de poco sirve que un widget envíe una señal si nadie es consciente de ello. Para ese propósito existen los receptores, conocidos como `slots`:

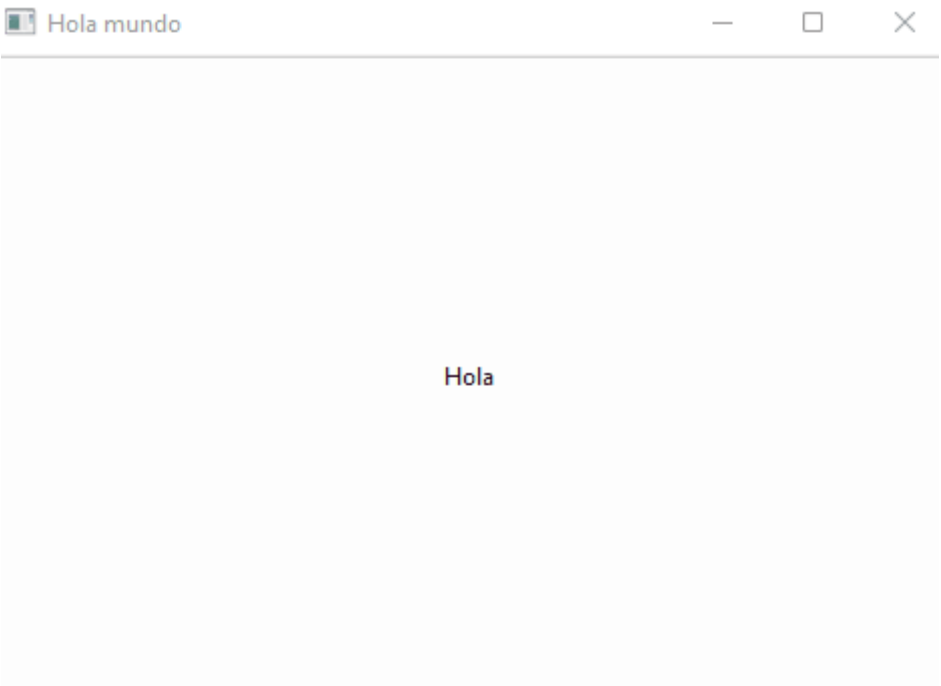
```
# Definimos un receptor para conectar la señal clicked a un método
button.clicked.connect(self.boton_clicado)

def boton_clicado(self):
    print("¡Me has clicado!")
```

Veamos otras señales de los botones para practicar:

```
# Pulsación y liberación
button.pressed.connect(self.boton_pulsado)
button.released.connect(self.boton_liberado)

def boton_pulsado(self):
    print("¡Me has pulsado!")

def boton_liberado(self):
    print("¡Me has liberado!")
# Señal para controlar el botón como un altenrador true/salse
button.setCheckable(True)
button.clicked.connect(self.boton_alternador)

def boton_alternador(self, valor):
    print("¿Alternado?", valor)
```

Con esto os podéis hacer una idea de cómo las señales están pendientes de todo lo que ocurre sobre los widgets para permitirnos actuar en consecuencia.

Componentes manipulables



Para acabar esta introducción vamos a ver cómo manipular un widget.

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:

```
# me gusta crear los accesos alfinal
self.button = button
```

Una vez contamos con el accesor, o mejor llamado puntero, podemos hacer referencia a cualquier instancia de un widget para modificarla a voluntad:

```
def boton_alternador(self, valor):
    if valor:
        self.button.setText("Estoy activado")
    else:
        self.button.setText("Estoy desactivado")
```

Hagamos un último ejemplo usando otro componente:

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


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

```
        super().__init__()
        self.setWindowTitle("Hola mundo")
        self.setMinimumSize(QSize(480, 320))

        # widget input de texto
        texto = QLineEdit()
        # capturamos la señal de texto cambiado
        texto.textChanged.connect(self.texto_modificado)

        # establecemos el widget central
        self.setCentralWidget(texto)

        # creamos el puntero
        self.texto = texto

    def texto_modificado(self):
        # recuperasmo el texto del input
        texto_recuperado = self.texto.text()
        # modificamos el título de la ventana al vuelo
        self.setWindowTitle(texto_recuperado)


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