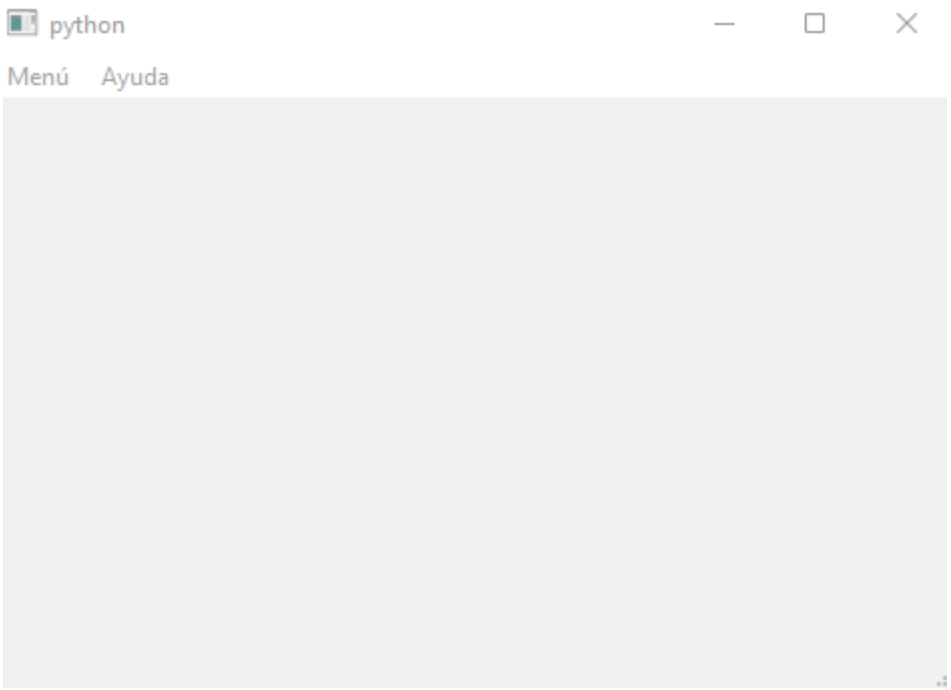
Barras de menú, estado y acciones QAction



Un menú es un componente estandar que se puede configurar en las ventanas principales `QMainWindow`.
Se ubican en la parte superior de la ventana o pantalla 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 accesiblidad:

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


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

        # 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 última acción
        menu_archivo.addAction("S&alir")


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

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 icono, un accesor y para que se llame el método `close` de la ventana principal:

```
from PySide6.QtGui import QIcon
from pathlib import Path

def absPath(file):
    return str(Path(__file__).parent.absolute() / file)


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

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:

```
from PySide6.QtWidgets import (QApplication, QMainWindow, QMessageBox)
# edited
from PySide6.QtGui import QAction, QIcon  # editado
from pathlib import Path
import sys

def absPath(file):
    return str(Path(__file__).parent.absolute() / file)

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

        menu = self.menuBar()
        menu_archivo = menu.addMenu("&Menú")
        menu_archivo.addAction("&Prueba")
        submenu_archivo = menu_archivo.addMenu("&Submenú")
        submenu_archivo.addAction("Subopción &1")
        submenu_archivo.addAction("Subopción &2")
        menu_archivo.addSeparator()
        menu_archivo.addAction(
            QIcon(absPath("exit.png")), "S&alir", self.close,
"Ctrl+Q")

        # Añadimos un menú de ayuda
        menu_ayuda = menu.addMenu("Ay&uda")
        # Creamos una acción específica para mostrar información
        accion_info = QAction("&Información", self)
        # Podemos configurar un icono en la acción
        accion_info.setIcon(QIcon(absPath("icon.png")))
```

```
        # También podemos especificar un accesor
        accion_info.setShortcut("Ctrl+I")
        # Le configuramos una señal para ejecutar un método
        accion_info.triggered.connect(self.mostrar_info)
        # Añadimos la acción al menú
        menu_ayuda.addAction(accion_info)

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

Las acciones también permiten configurar lo que se conoce como `statusTip` para mostrar la utilidad de la acción.

```
# Añadimos un texto de ayuda
accion_info.setStatusTip("Muestra información irrelevante")
```

Esto no hará nada porque el texto se muestra en la barra de estado de la ventana principal, una barra que no tenemos activa. Así que vamos a darla de alta:

```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QMessageBox, QStatusBar)  # edited

# Añadimos una barra de estado
self.setStatusBar(QStatusBar(self))
```

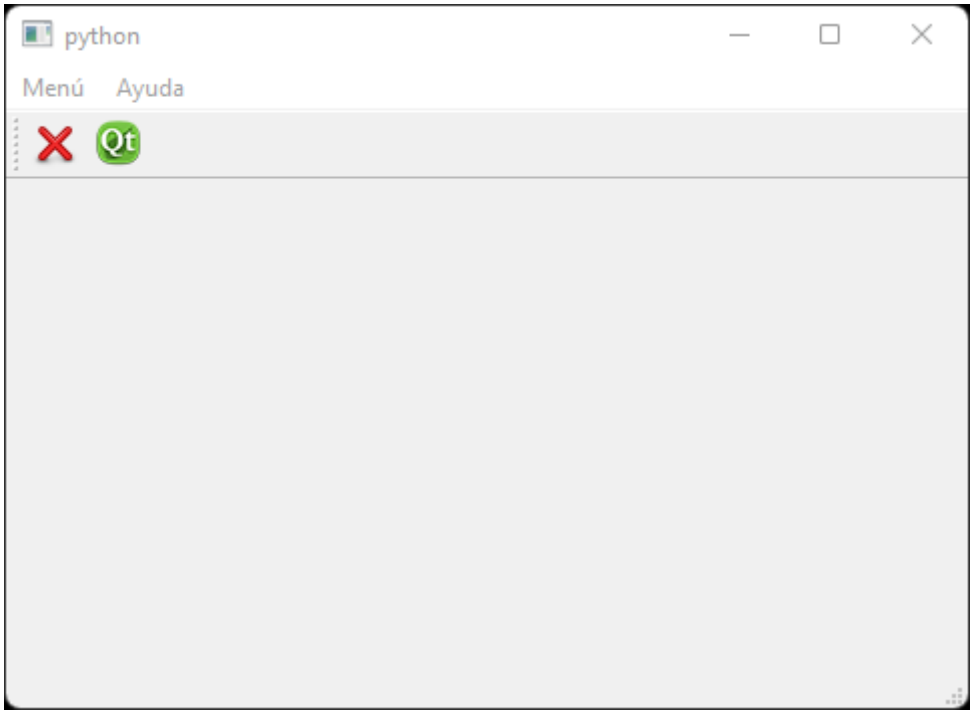
Ahora al pasar el ratón por encima de la acción aparecerá en la parte inferior el texto explicativo.

En la siguiente lección seguiremos mejorando esta ventana principal, pero antes vamos a refactorizar un poco el código para aligerar el constructor:

```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QMessageBox, QStatusBar)
from PySide6.QtGui import QAction, QIcon
from pathlib import Path
import sys

def absPath(file):
    return str(Path(__file__).parent.absolute() / file)

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

        # 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")

        # Añadimos un menú de ayuda
        menu_ayuda = menu.addMenu("Ay&uda")
        # Creamos una acción específica para mostrar información
        accion_info = QAction("&Información", self)
        # Podemos configurar un icono en la acción
        accion_info.setIcon(QIcon(absPath("info.png")))
        # También podemos especificar un accesor
        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)

        # Añadimos una barra de estado
        self.setStatusBar(QStatusBar(self))

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

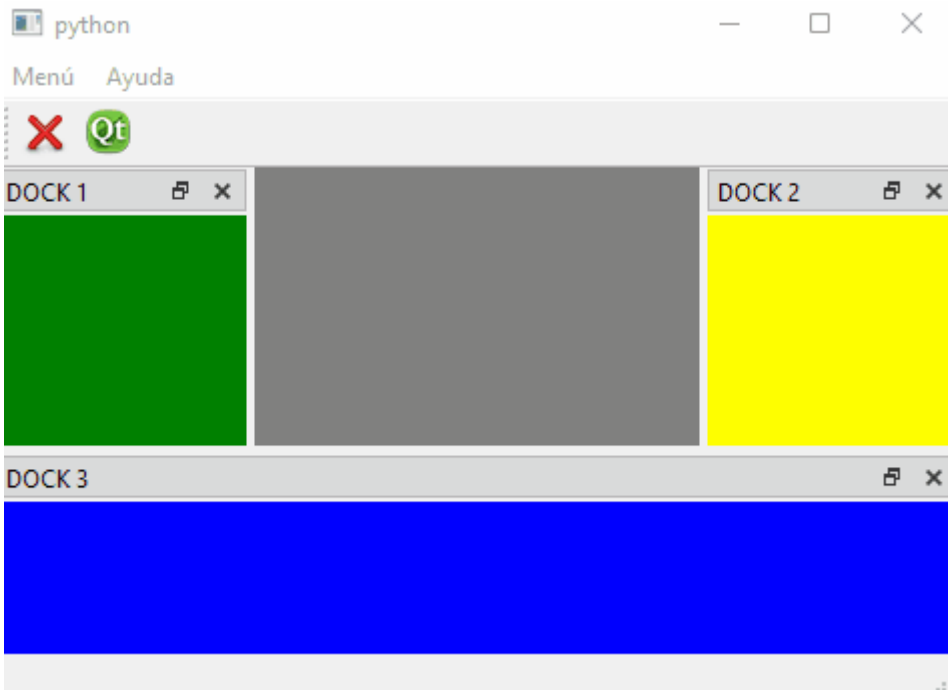
Barra de herramientas QToolBar



Las barras de herramientas son otros componentes estandar 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:

```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QMessageBox, QStatusBar, QToolBar)  #
edited
from PySide6.QtGui import QAction, QIcon
from pathlib import Path
import sys

def absPath(file):
    return str(Path(__file__).parent.absolute() / file)

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.resize(480, 320)
        self.construir_menu()
        # construimos las herramientas
        self.construir_herramientas()

    def construir_menu(self):
        # accesores de clase
        self.accion_info = accion_info

    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("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)

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

Docks flotantes QDockWidget



El último componente que nos falta ver de las ventanas principales son los docks floatantes.

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 tiene un método `setWidget` para configurar el widget que contendrán. Nuestra clase `Caja` es muy simple y nos permitirá hacernos una idea del funcionamiento, así que vamos a recuperarla para utilizarla de widget de los docks:

```
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QMessageBox, QStatusBar,
    QToolBar, QLabel, QDockWidget)  # edited

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

class MainWindow(QMainWindow):
    def __init__(self):
        # ...
        # añadimos los docks
        self.construir_docks()
        # creamos una caja como widget central de la ventana principal
        self.setCentralWidget(Caja("gray"))

    def construir_docks(self):
        # creamos un dock
        dock1 = QDockWidget()
        # le damos un título (optativo)
        dock1.setWindowTitle("DOCK 1")
        # establecemos el widget que contendrá
        dock1.setWidget(Caja("green"))
```

```
        # ancho mínimo (optativo)
        dock1.setMinimumWidth(100)
        # lo añadimos en una posición de la ventana principal
        self.addDockWidget(Qt.LeftDockWidgetArea, dock1)


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

Como véis los `docks` son super flexibles. Eso sí, al cerrarlo lo perdemos y deberíamos proveer de alguna forma de crearlo de nuevo, o también podemos limitar sus características :

```
dock1.setFeatures(
    QDockWidget.NoDockWidgetFeatures | QDockWidget.DockWidgetFloatable
|
    QDockWidget.DockWidgetClosable | QDockWidget.DockWidgetMovable)
```

También podemos controlar su tamaños:

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

Y añadir más docks en otras posiciones para juguetear con ellos:

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

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

Fijaros como podemos acoplar widgets unos sobre otrossi hay suficiente espacio o apilarlos usando pestañas, dan muchísimo juego.