PRÁCTICAS

Ingeniería de Sistemas Software Basados en Conocimiento

Gonzalo Márquez de Torres

Curso 2022-23

ÍNDICE

PRÁCTICA I. BASES PARA LA IMPLEMENTACIÓN DE UNA ARQUITECTURA MODELO VISTAS-CONTROLADOR

PRÁCTICA I. BASES PARA LA IMPLEMENTACIÓN DE UNA ARQUITECTURA MODELO VISTAS-CONTROLADOR

1. Instalación y primeros pasos en Spyder.

En primer lugar, he comenzado la instalación de Spyder, mediante la línea de comandos con *sudo apt-get install spyder*. Una vez, instalado he comprobado que funciona correctamente inicializando la aplicación y pegando un código de ejemplo. Además, también, he incluido la biblioteca gráfica Qt para el lenguaje de programación Python por medio del comando *sudo apt-get install pyqt5-dev-tools.*

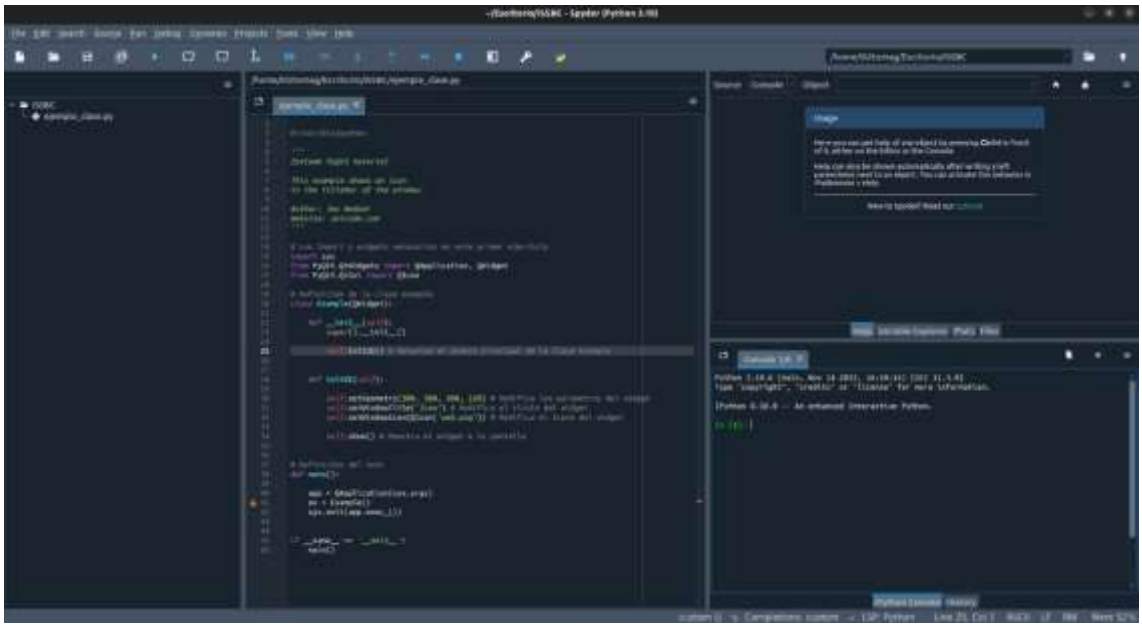

Imagen 1.1.1. Interfaz de Spyder.

Tras la instalación, comenzamos con el segundo paso, la familiarización con el entorno de desarrollo. En primer lugar, crearemos un nuevo proyecto, en una carpeta creada para esta asignatura, coloco la ubicación donde quiero ubicarla, tras finalizar esta acción aparecerán todos los archivos de la carpeta seleccionada.

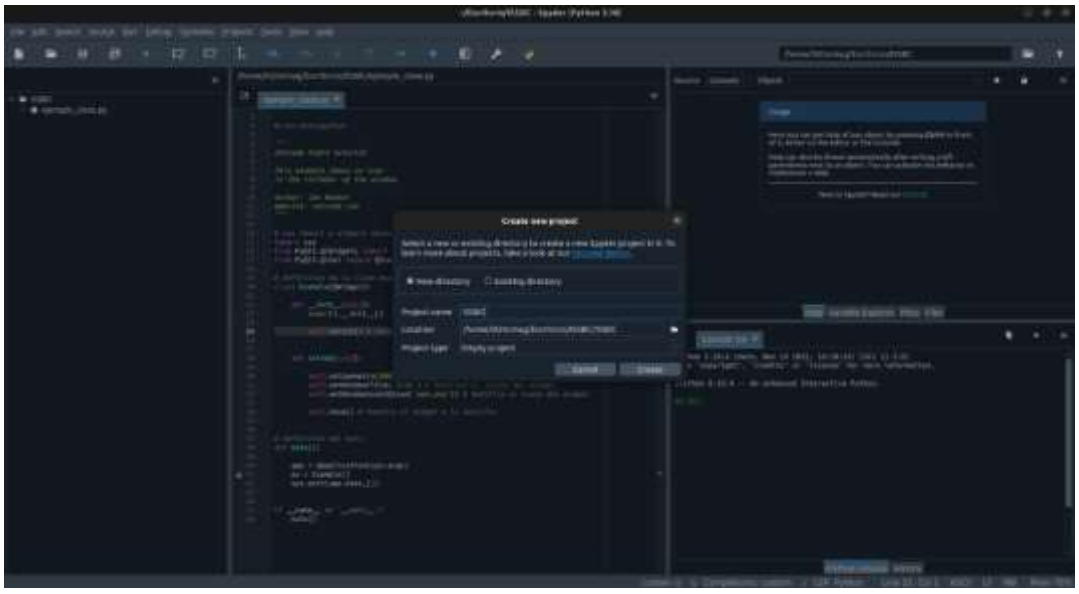

Imagen 1.1.2. Creación de un proyecto en Spyder.

Tras la creación del proyecto, pasaremos a la modificación de los paneles en el menú de *"Ver/View",* y dejaremos activadas solo Editor *(editor),* Terminal de Python *(iPython Console),* Archivos *(files)* y Proyecto *(project)* y deshabilitando el resto.

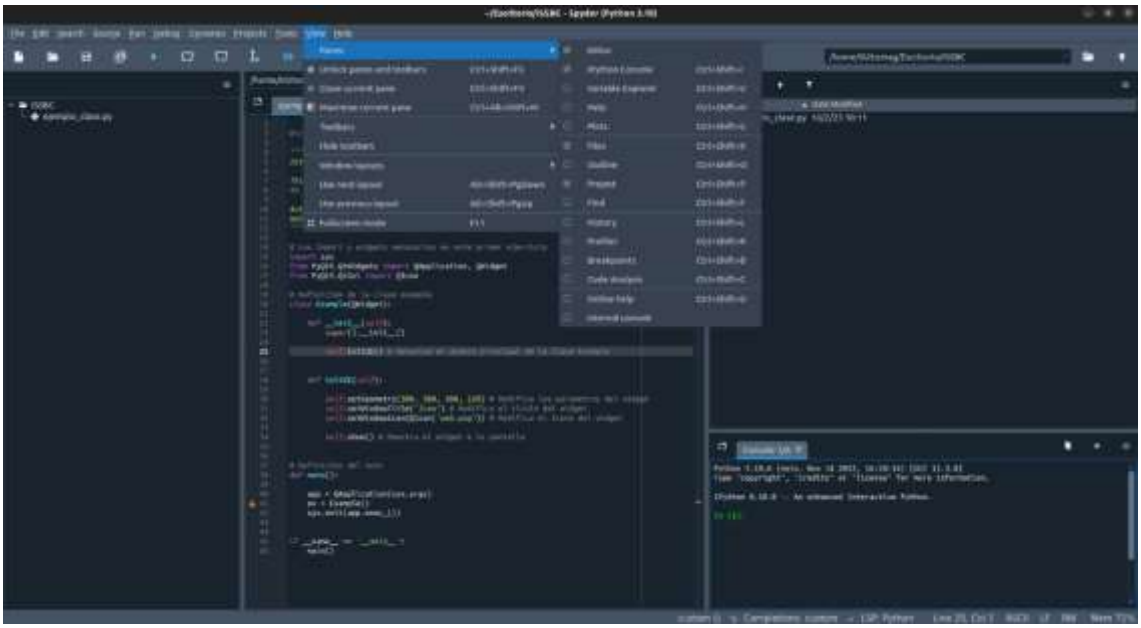

Imagen 1.1.3. Modificación en el menú de Vistas en Spyder.

Otro dato curioso, es que se podrá crear un archivo Python desde este menú, accediendo a Crear Nuevo *(New file...)* y se crea por defecto un nuevo archivo con formato Python, el cual guardaremos con un nombre determinado y aparecerá en la barra de navegación del proyecto, siempre que la guardemos en la carpeta seleccionada en el paso previo.

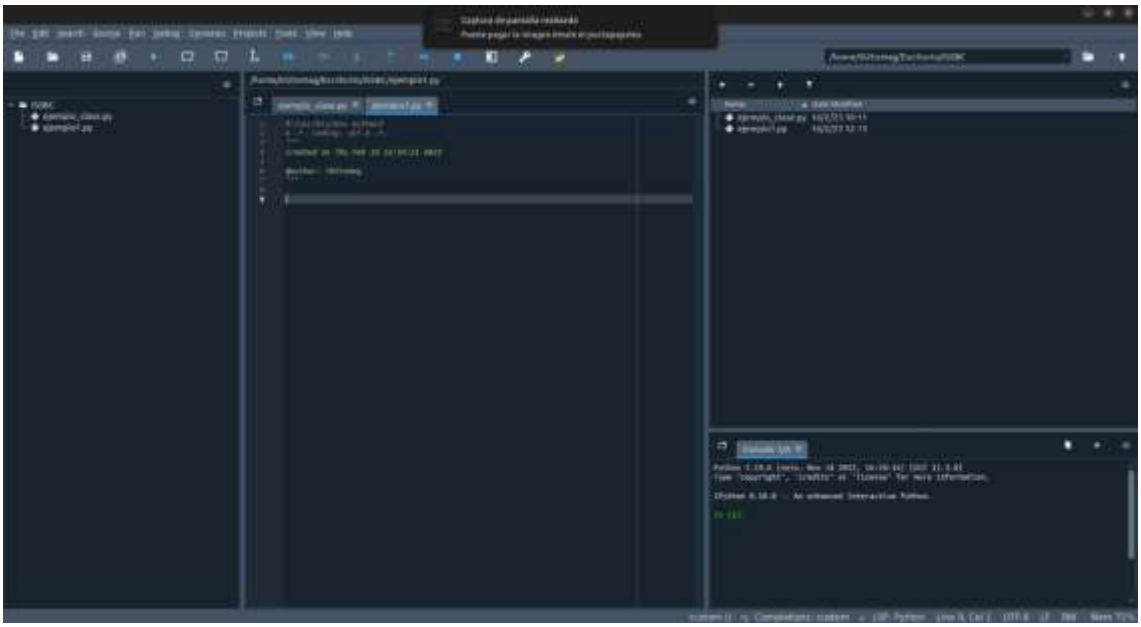

Imagen 1.1.4. Creación de un fichero Python en Spyder.

2. Bases PYQT.

Para introducirnos en PYQT, accederemos a un tutorial para comenzar las nociones básicas (*https://zetcode.com/qui/pyqt5/*), procediendo a analizar los diferentes métodos y funciones que se utilizarán, prestando atención en aquellos aspectos importantes como gestores de eventos o la creación de ventanas emergentes. Las clases de *PyQt5* se dividen en varios módulos, incluidos los siguientes:

- El módulo *QtCore* contiene la funcionalidad principal, permitiendo trabajar con el tiempo, los archivos y directorios, varios tipos de datos...
- *QtGui* contiene clases para la integración de sistemas de ventanas, manejo de eventos, gráficos 2D, imágenes básicas, fuentes y texto.
- El módulo *QtWidgets* permite crear interfaces de usuario de estilo de escritorio clásico.
- QtMultimedia soporta manejar contenido multimedia y API para acceder a la funcionalidad de cámara y radio.
- *QtBluetooth* tolera buscar dispositivos y conectarse e interactuar con ellos.
- El módulo *QtNetwork* contiene las clases para la programación de redes, facilitando la codificación de clientes y servidores TCP/IP y UDP (*portabilidad*).
- *QtPositioning* contiene clases para determinar una posición variando el uso de fuentes posibles.
- El módulo *Enginio* implementa una biblioteca para acceder al tiempo de ejecución de aplicaciones administradas de *Qt Cloud Services*.
- El módulo *QtWebSockets* contiene clases que implementan el protocolo *WebSocket*.
- *QtWebEngineCore* contiene las clases principales de *Web Engine*.
- *QtWebEngineWidgets* contiene el navegador web basado en *Chromium*.
- *QtXml* contiene clases para trabajar con archivos XML, pudiendo ser implementado para API SAX y DOM.
- El módulo *QtSvg* proporciona clases para mostrar el contenido de los archivos SVG (*Scalable Vector Graphics*), un lenguaje para describir gráficos bidimensionales y aplicaciones gráficas en XML.
- El módulo *QtSql* proporciona clases para trabajar con bases de datos.
- *QtTest* contiene funciones que permiten la prueba unitaria de las aplicaciones *PyQt5*.

2.1. Fecha y Hora

En el ámbito de la fecha y hora, PyQt5 tiene diferentes métodos para determinar la fecha y hora actuales como *currentDate*, devuelve fecha actual, *currentTime* devuelve hora actual y currentDateTime, devuelve fecha y hora actual. Para visualizar la fecha que queremos mostrar, habrá diferentes formatos en función de nuestras preferencias:

```
# Asocia a la variable now la fecha actual
now = QDate.currentDate() # Devuelve fecha actual

# Muestra la fecha actual del sistema en diferentes formatos
print(now.toString(Qt.ISODate))  # YY-MM-dd
print(now.toString(Qt.DefaultLocaleLongDate)) ## dia, dd de MM de YY

datetime = QDateTime.currentDateTime() # Devuelve fecha y hora actual
print(datetime.toString()) # dia mes dd HH:MM:SS YY

time = QTime.currentTime() # Devuelve hora actual
print(time.toString(Qt.DefaultLocaleLongDate)) # HH:MM:SS
```

Imagen 1.2.1. Diferentes funciones para mostrar fechas en el programa.

Debido a los diferentes horarios en función de la zona geográfica donde nos encontremos, se utiliza UTC. Luego, podremos usar la función *toLocalTime* para convertir una hora universal en la hora local correspondiente a nuestro UTC. Además, contaremos con la función *daysInMonth* devuelve el número de días de un mes en particular y el método *daysInYear* devuelve el número de días que han pasado entre dos fechas. Existe otra función, *daysTo* que devuelve el número de días desde una fecha hasta otra.

```
#DIAS Y MES
xmas1 = QDate(2019, 12, 24) # Asocia la fecha 24-12-2019 a xmas1
xmas2 = QDate(2020, 12, 24) # Asocia la fecha 24-12-2020 a xmas2

dayspassed = xmas1.daysTo(now)
print(f'{dayspassed} dias han pasado desde el ultimo XMas')

nofdays = now.daysTo(xmas2)
print(f'Quedan {nofdays} dias hasta la proxima XMas\n') # Da un resultado negativo,
```

Imagen 1.2.2. Uso de las funciones en daysTo y dayspasses en el programa.

```
Fecha:  2023-02-18
Fecha:  sábado, 18 de febrero de 2023
Fecha y Hora:  sáb feb 18 11:25:12 2023
Hora:  11:25:12 (CET)


1152 dias han pasado desde el ultimo XMas
Quedan -786 dias hasta la proxima XMas
```

Imagen 1.2.3. Salida por pantalla de la Imagen 1.2.2.

Se podrá acceder al código por medio de este enlace http://www.uco.es/~i02matog/practica1/p1_ejemplo1_fechas.txt.

2.2. Creación de Ventanas y Botones

Los widgets básicos se encuentran en el módulo *PyQt5.QtWidgets*. El parámetro *sys.argv* es una lista de argumentos de una línea de comando. Un dato importante es que los scripts de *Python* se pueden ejecutar desde el terminal, siendo una forma de controlar el inicio de estos.

El widget *QWidget* es la clase base de todos los objetos de interfaz de usuario en *PyQt5*. El constructor predeterminado para *QWidget* no tiene padre y es conocido como ventana. Existirán diferentes métodos para modificar diferentes parámetros de la ventana, como, por ejemplo, *resize* permite modificar el tamaño del widget, el título de la ventana se podrá cambiar con *setWindowTitle*, el cual se muestra en la barra de título de la misma ventana. Además, también podremos añadir un determinado icono (*imagen pequeña que se muestra en la esquina superior izquierda junto al título*), con el método *setWindowIcon*. Por último, el método *show* muestra el widget en la pantalla, el cual se crea primero en la memoria y luego se muestra en la pantalla. El código será accesible en http://www.uco.es/~i02matog/practica1/p1_ejemplo2_firstprograms.txt.

El bucle principal recibe eventos del sistema de ventanas y los envía a los widgets de la aplicación. El bucle principal finaliza si llamamos al método de salida o se destruye el widget principal. El método *sys.exit* asegura una salida limpia, informando al medio cómo finalizó la aplicación.

Ahora creamos una nueva clase llamada *Example*, la cual hereda de la clase *QWidget*. Esto significa que llamamos a dos constructores: el primero para la clase principal y el segundo para la clase heredada. El método super devuelve el objeto principal de la clase *Exemple* y llamamos a su constructor. El método *__init__* es un método constructor en lenguaje Python. La creación de la GUI se delega al método *initUI*.

```
class Example(QWidget):

    def __init__(self):
        super().__init__()
        self.initUI()


    def initUI(self):
        QToolTip.setFont(QFont('Calibri', 12)) # Modifica el tipo de letra y tamaño

        self.setToolTip('This is a <b>QWidget</b> widget') # Crea información sobre la herramienta

        btn = QPushButton('Cerrar', self) # Creación del boton
        btn.clicked.connect(QApplication.instance().quit)
        btn.setToolTip('This is a <b>QPushButton</b> widget') # Crea información sobre la herramienta
        btn.resize(btn.sizeHint()) # Tamaño idoneo predefinido
        btn.move(50, 50) # Determinar el tamaño de la ventana

        self.setGeometry(300, 300, 300, 200) # Geometria de la ventana
        self.setWindowTitle('Ejemplo Botones') # Titulo de la ventana
        self.show() # Mostrar ventana


def main():

    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())


if __name__ == '__main__':
    main()
```

Imagen 1.2.4. Programa para crear una ventana modificando algunos parámetros.

Este programa será accesible por medio del siguiente enlace, http://www.uco.es/~i02matog/practica1/p1_ejemplo3_botones1.txt.

El método *setFont* permite modificar la fuente y el tamaño de la letra para representar información sobre herramientas. Para crear una información sobre herramientas, llamamos al método *setTooltip*.

Otro dato curiosos, es que podremos crear widgets de botón pulsador, modificando el tamaño y la posición dentro de la ventana creada, aunque tendremos el método *sizeHint* que proporciona un tamaño recomendado para el botón. Además, se puede insertar un determinado texto en el botón mediante *QPushButton*. A la hora de cerrar una ventana, deberemos introducir un botón con los métodos aprendidos y se utilizará *clicked.connect* asociándolo a la función *quit* para proceder al cierre de la pestaña.

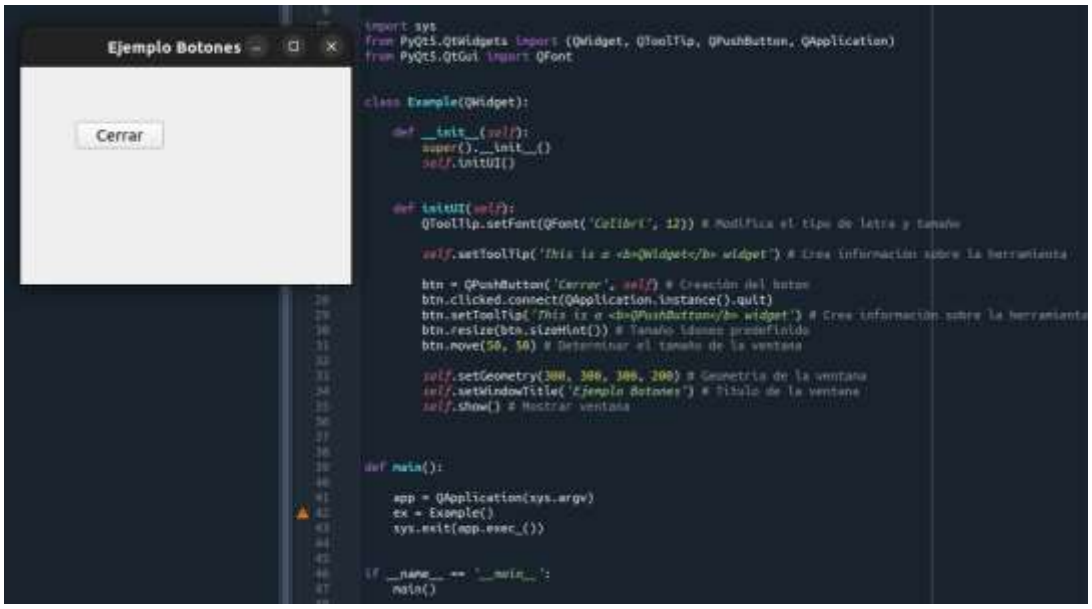

Imagen 1.2.5. Creación de botón con la acción de cerrar la pes.

En ocasiones, querremos modificar el comportamiento del botón creado. Por ejemplo, si tenemos un archivo abierto en un editor al que le hicimos algunos cambios, luego, queremos informar al cliente si desea salir. En este ejemplo, si hace clic en un botón Sí, aceptamos el evento, es decir, el cierre del widget y la finalización de la aplicación, pero, de lo contrario, ignoramos el evento de cierre y continuamos en la ventana. El código se encontrará en el siguiente link, http://www.uco.es/~i02matog/practica1/p1_ejemplo4_botones2.txt

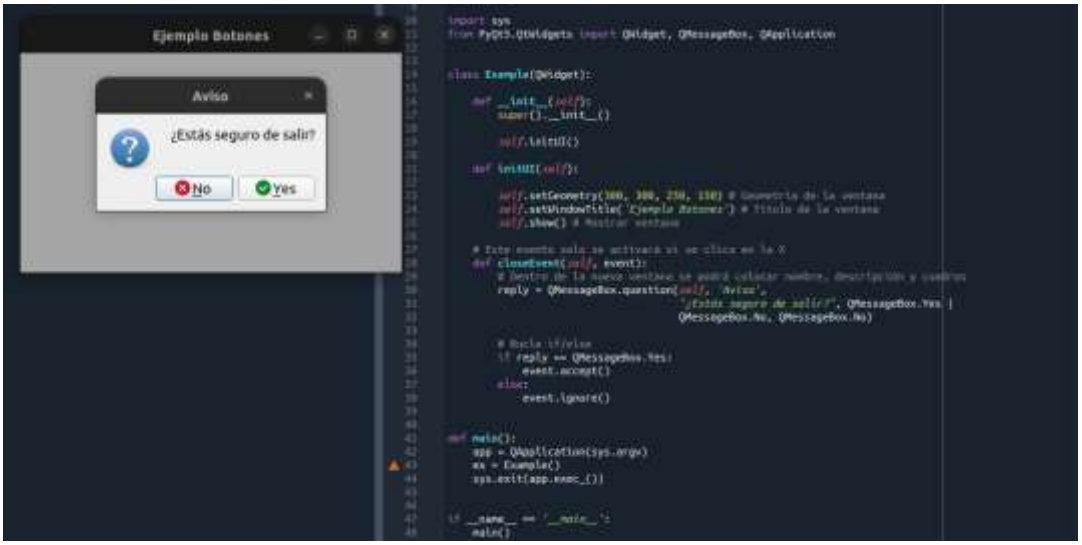

Imagen 1.2.6. Creación de un aviso ante el cierre de la ventana.

Por otro lado, en cuanto al diseño de la ventana, habrá diferentes funciones que se encargarán de centrar diferentes elementos en el centro de la pantalla:

- El método *self.center()* que centrará la ventana.
- Con *self.frameGeometry()* obtenemos un rectángulo que especifica la geometría de la ventana principal, incluyendo el marco de ventana.
- Con *cp=QDesktopWidget().AvailableGeometry().center(),* se calcula la resolución de pantalla del monitor, y a partir de esta resolución, obtenemos el punto central.
- Con *qr.moveCenter(cp)* nuestro rectángulo ya tiene su ancho y alto, luego se coloca el centro del rectángulo en el centro de la pantalla. Cabe decir, que el tamaño del rectángulo no cambia.
- Con *self.move(qr.topLeft())* movemos el punto superior izquierdo de la ventana de la aplicación al punto superior izquierdo del rectángulo *qr*, centrando así la ventana en nuestra pantalla.

2.3. Eventos y Señales.

Las aplicaciones GUI están dirigidas por eventos, los cuales son generados principalmente por el usuario de una aplicación. Aunque también pueden generarse por otros medios (*una conexión a Internet, un administrador de ventanas o un temporizador*). Cuando llamamos al método *exec_* de la aplicación, la aplicación ingresa al ciclo principal. El bucle principal obtiene eventos y los envía a los objetos. En el modelo de evento, hay tres participantes:

- Origen del evento: objeto cuyo estado cambia y genera eventos.
- Objeto de evento: encapsula los cambios de estado en la fuente del evento.
- Destino del evento: objeto que desea recibir una notificación.
- Objetivo del evento: delega la tarea de manejar un evento al destino del evento.

PyQt5 tiene un mecanismo único de señal y ranura para manejar eventos. Las señales y las ranuras (*cualquier objeto de Python*) se utilizan para la comunicación entre objetos, como se puede observar en el siguiente enlace del código,

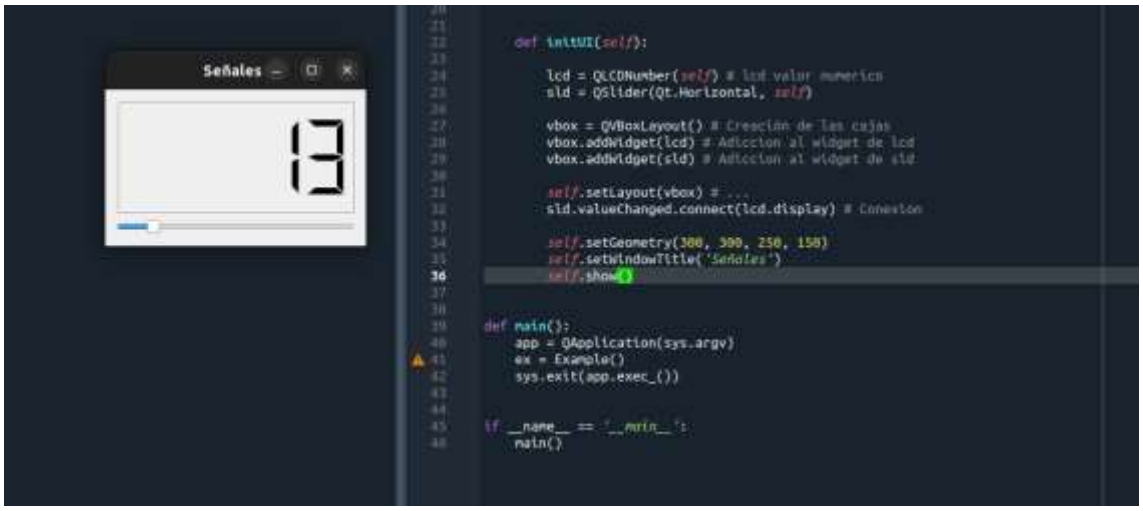

Imagen 1.2.7. Creación de un evento que modifica su valor a medida que avanza.

http://www.uco.es/~i02matog/practica1/p1_ejemplo11_senales.txt. Se emite una señal cuando ocurre un evento en particular. Se llama a una ranura cuando se emite su señal conectada.

En el ejemplo, mostramos un *QtGui.QLCDNumber* y un *QtGui.QSlider*. Cambiamos el número de *lcd* arrastrando la pestaña deslizante. Con una señal *valueChanged* del control deslizante variará el número de lcd.

El emisor es un objeto que envía una señal y el receptor es el objeto que recibe la señal. La ranura es el método que reacciona a la señal. Los eventos en PyQt5 se procesan a menudo mediante la reimplementación de los controladores de eventos. En nuestro ejemplo, introducimos el controlador de eventos *keyPressEvent*. Si hacemos pulsamos la tecla botón Escape, la aplicación finaliza.

```
def keyPressEvent(self, e):
    if e.key() == Qt.Key_Escape:
        self.close()
```

Imagen 1.2.8. Función para que al pulsar Escape cierre la pestaña.

El objeto de evento es un objeto de *Python* que contiene una serie de atributos que describen el evento. El objeto de evento es específico del tipo de evento generado. Las coordenadas x e y se muestran en un widget *QLabel*.

El seguimiento del ratón se podrá activar con *setMouseTracking(true)*, ya que está deshabilitado de forma predeterminada. Por lo que el widget solo recibe eventos de movimiento del cuando se mueve el ratón por la ventana, incluso si no se presiona ningún botón.

El objeto de evento contiene datos sobre el evento que se desencadenó. En nuestro caso, un evento de movimiento del ratón. Con los métodos x e y determinamos las coordenadas x e y de la ventana del puntero del ratón. A veces conviene saber qué widget es el emisor de una señal. Para esto, PyQt5 tiene el método del remitente.

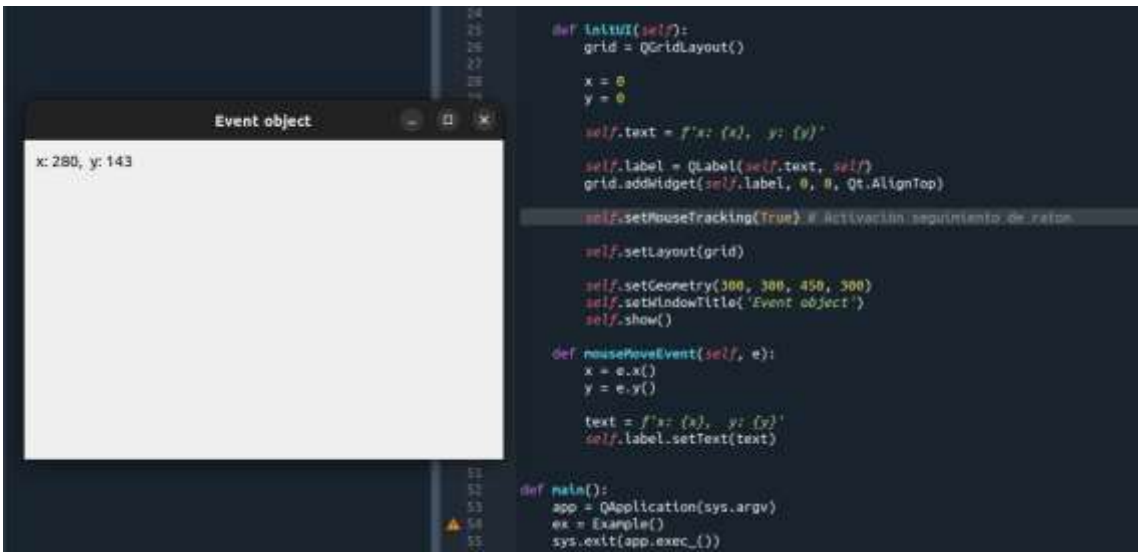

Imagen 1.2.9. Programa para ver la coordinada x e y donde se encuentra el ratón.

El código del ejemplo anterior se podrá observar en el siguiente enlace, http://www.uco.es/~i02matog/practica1/p1_ejemplo12_mapapixel.txt.

Para el siguiente ejemplo, utilizaremos dos botones, a los cuales asociaremos el método *buttonClicked* al cual irá asociado una determinada acción cada vez que hagamos click en el botón. Ambos botones están conectados a un mismo evento *clicked.connect(self.buttonClicked)*, es decir, ambos botones están conectados a la misma ranura. En la barra de estado de la aplicación, mostramos la etiqueta del botón que se está presionando, a través del siguiente código, http://www.uco.es/~i02matog/practica1/p1_ejemplo13_botones.txt.

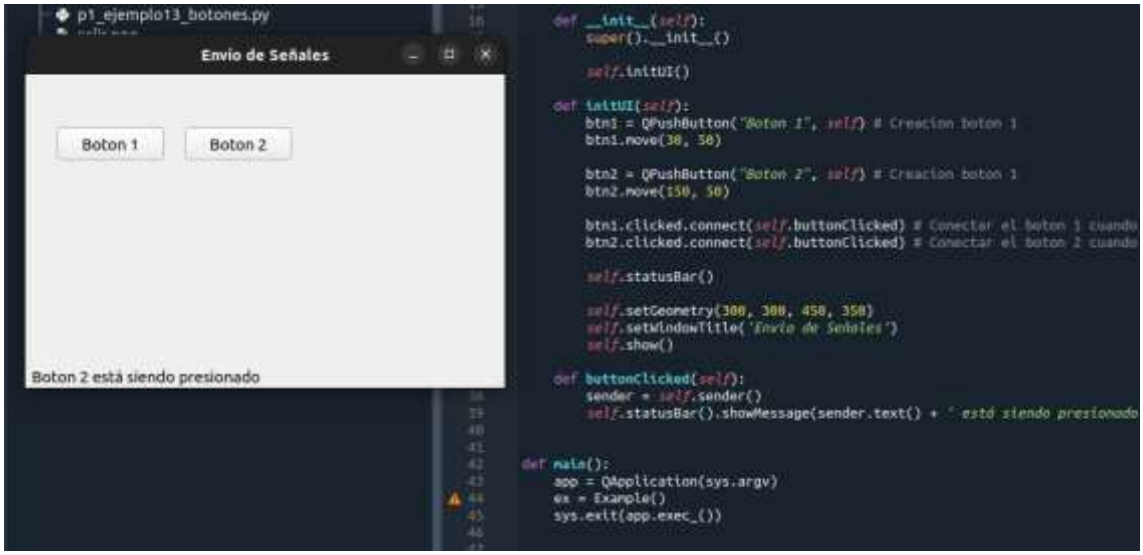

Imagen 1.2.10. Programa que activa un mensaje cuando se presiona un botón.

Los objetos creados a partir de un *QObject* pueden emitir señales. El siguiente ejemplo muestra cómo emitir señales personalizadas. Creamos una nueva señal llamada *closeApp*. Esta señal se emite durante un evento de presión del ratón. La señal se conecta a la ranura de cierre de *QMainWindow*. Se crea una señal con *pyqtSignal* como un atributo de clase de la clase *Communicate* externa. La señal personalizada de *closeApp* está conectada a la ranura de cierre de *QMainWindow*. Cuando hacemos clic en la ventana con el puntero del ratón, se emite la señal *closeApp* y la aplicación termina.

3. Menú y Barra de Herramientas en Aplicaciones.

La clase *QMainWindow* proporciona una ventana de aplicación principal, permitiendo crear un esqueleto de aplicación clásico con una barra de estado (*muestra información sobre el estado de la ventana*), de herramientas y de menú. La barra de estado se crea con la ayuda del widget *QMainWindow*. Para obtener la barra de estado, llamamos al método *statusBar* de la clase *QtGui.QMainWindow*. La primera llamada del método crea una barra de estado. Las llamadas posteriores devuelven el objeto de la barra de estado. *showMessage* muestra un mensaje en la barra de estado.

3.1. Barra de Menú.

Una barra de menú, parte común de una aplicación GUI, es un grupo de comandos ubicados en varios menús. Para la primera prueba, crearé un menú que contenga una acción que finaliza la aplicación si se selecciona. Además, añadiremos la acción del atajo de teclado (*Ctrl+Q*), mediante una serie de llamadas:

- El método *QAction* permite definir el nombre de la acción dentro del menú y añadir un icono a esta.
- El uso de setShortcut permite indicar un comando indicado para realizar dicha acción, es decir, añadir un atajo de teclado a dicha acción.
- Con *setStatusTip* añade información adicional a la acción cuando pasamos el ratón sobre el elemento del menú.

Cuando seleccionamos la acción *triggered.connect(qApp.quit))*, se emite una señal disparada. La señal está conectada al método de salida del widget *QApplication*. Esto finaliza la aplicación. Con los siguientes comandos se procederá a crear la barra del menú.

- El uso de *menuBar()* permite crea una barra de menú.
- El método *addMenu('&Archivo')* añade un elemento dentro del menú con un determinado nombre.
- Con *addAction(exitAction)* añadimos una acción determinada, en el caso del ejemplo introducido, salir de la ventana.

También podremos crear un submenú dentro de un menú, simplemente en lugar de añadir una acción, añadiríamos el submenú por medio de *QMenu*, luego el nuevo submenú se crearía mediante *QMenu*, como se puede ver en el siguiente link, http://www.uco.es/~i02matog/practica1/p1_ejemplo5_menu.txt.

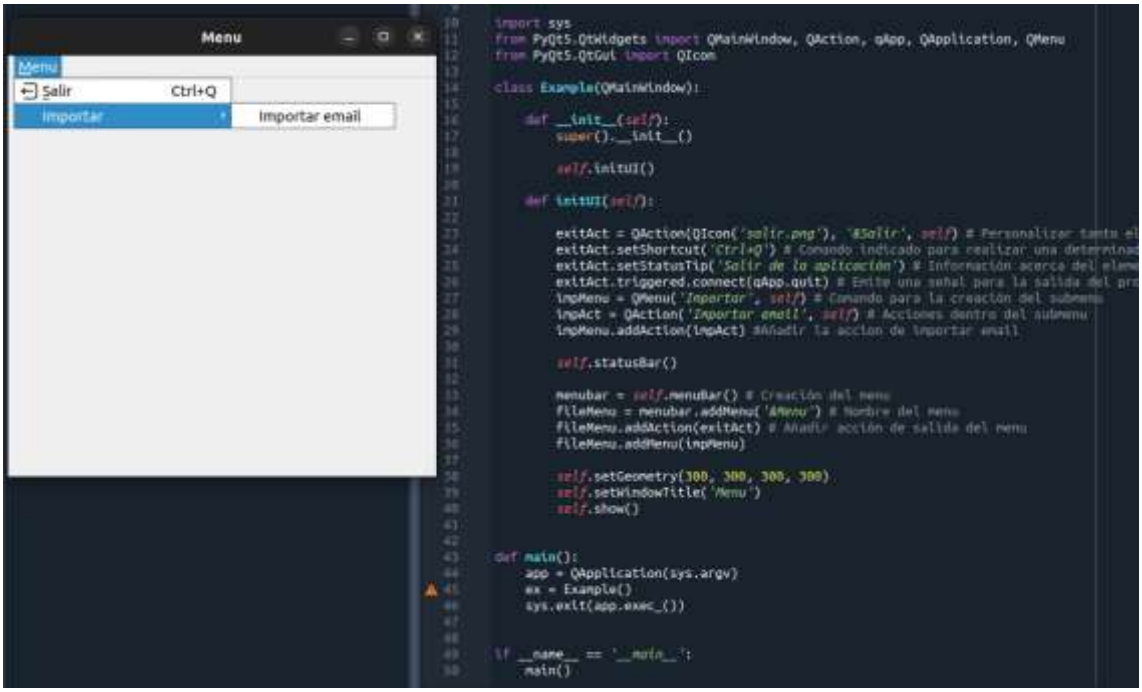

Imagen 1.3.1. Creación de un menú incluyendo la acción de salir y un submenú (importar).

Otra opción dentro de este menú será indicar una determinada selección, como ver la barra de estado (*mostrar/ocultar esta barra*). Cuando la barra de estado está visible, el elemento del menú estará marcado. Con la opción *checkable* dentro de la función *QAction* creamos un menú seleccionable. Posteriormente habrá que indicar que ocurre cuando se selecciona esta opción:

- Con *setChecked(true)* la barra de estado es visible desde el principio, verificamos la acción con el método *toggleMenu(self, estado)*.
- Si el estado es verdadero, mostramos la barra mediante *statusbar.show()*.
- En caso contrario, utilizamos *statusbar.hide()*, para ocultar la barra de estado.
- Será accesible por medio de http://www.uco.es/~i02matog/practica1/p1_ejemplo6_barraherramientas.txt.

Imagen 1.3.2. Creación de una opción seleccionable.

Un menú contextual, también llamado menú emergente, es una lista de comandos que aparece en algún contexto. Por ejemplo, en un navegador web Opera cuando hacemos clic derecho en una página web, obtenemos un menú contextual, donde podremos realizar diferentes acciones como actualizar, retroceder...

Si hacemos clic derecho en la ventana, obtenemos un menú contextual. Para insertar esta acción, deberemos implementar el método *mapToGlobal*. El menú contextual se muestra con el método *exec_*, el cual obtiene las coordenadas del puntero del mouse del objeto de evento. Con el método *mapToGlobal* traduce las coordenadas del widget a las coordenadas de la pantalla global. Si la acción devuelve desde el menú igual a la acción de salir (*qApp*), terminamos la aplicación.

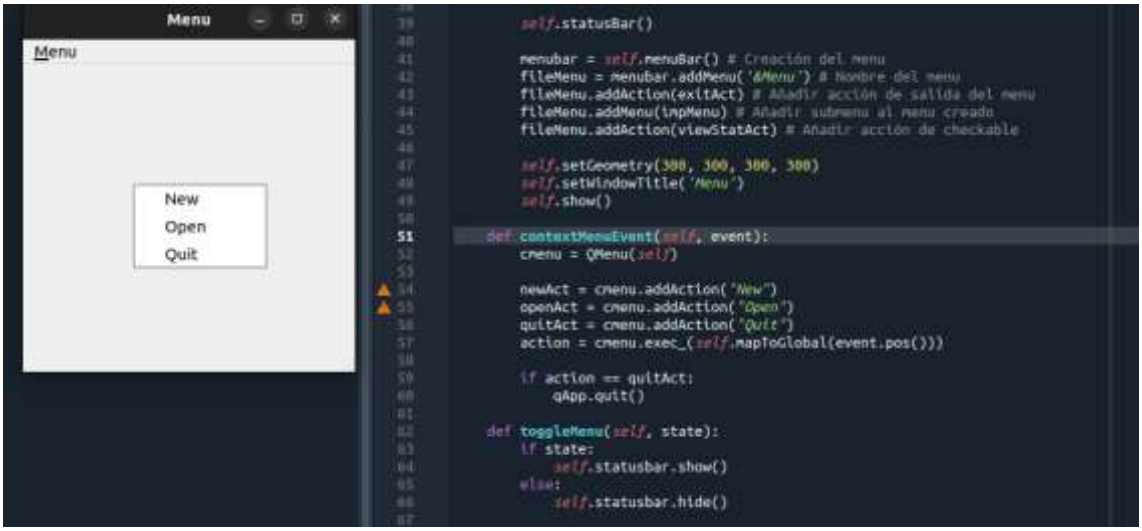

Imagen 1.3.3. Creación de un menú incluyendo la acción de salir y un submenú (importar).

Imagen 1.3.4. Creación de un menú incluyendo la acción de salir y un submenú (importar).

Los menús agrupan todos los comandos que podemos utilizar en una aplicación, en cambio, las barras de herramientas proporcionan un acceso rápido a los comandos más utilizados, teniendo acciones de herramienta o de salida que finaliza la aplicación cuando se activa. El objeto tiene una etiqueta, un icono y un acceso directo. La barra de herramientas se crea con el método *addToolBar* y se agrega un objeto de acción con *addAction*. Se podrá acceder al código por medio de la siguiente página, http://www.uco.es/~i02matog/practica1/p1_ejemplo7_menu_barraherramientas.txt.

Además, se podrá combinar el menú de herramientas con el menú y el menú contextual para dar paso a una ventana más completa.

3.2. Gestión de diseño en PyQt5.

La gestión del diseño es la forma en que colocamos los widgets en la ventana de la aplicación, pudiendo utilizarse un posicionamiento absoluto o con clases de diseño. Con respecto al posicionamiento absoluto, el programador especifica la posición y el tamaño de cada widget en píxeles, teniendo las siguientes limitaciones:

- El tamaño y la posición de un widget no cambian si redimensionamos una ventana.
- Las aplicaciones pueden verse diferentes en varias plataformas.
- Cambiar las fuentes en nuestra aplicación podría estropear el diseño
- Si decidimos cambiar nuestro diseño, debemos rehacerlo por completo, lo cual es laborioso y requiere mucho tiempo.

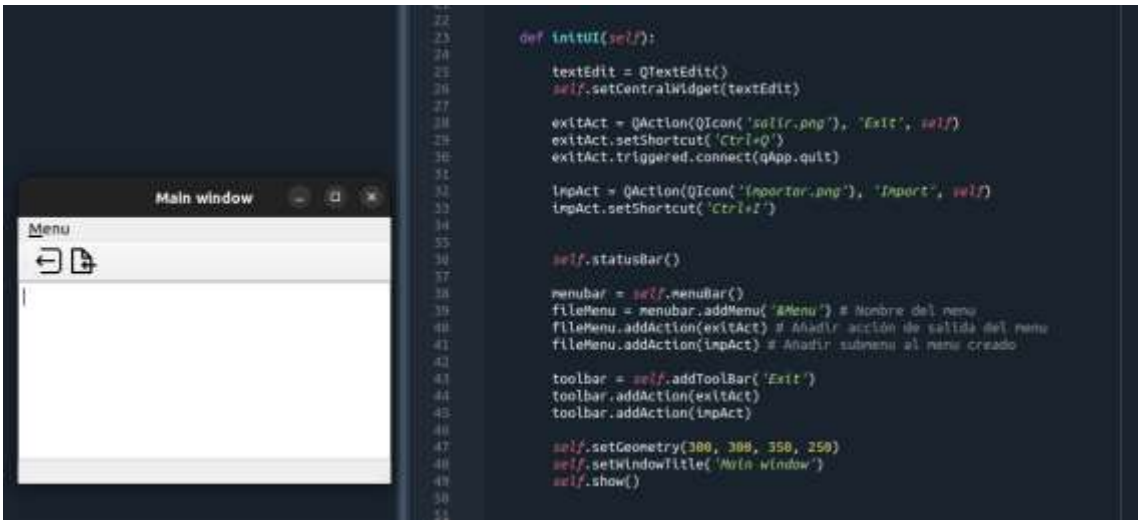

Imagen 1.3.5. Creación de una ventana con menú y barra de herramientas.

Normalmente, utilizamos el método *move* para posicionar nuestros widgets. En nuestro caso, el texto se introducirá por medio de etiquetas y se posicionan en unas coordenadas *x* e *y*. El comienzo del sistema de coordenadas está en la esquina superior izquierda. Los valores de x crecen de izquierda a derecha. Los valores de y crecen de arriba hacia abajo, como se puede ver http://www.uco.es/~i02matog/practica1/p1_ejemplo8_layout.txt.

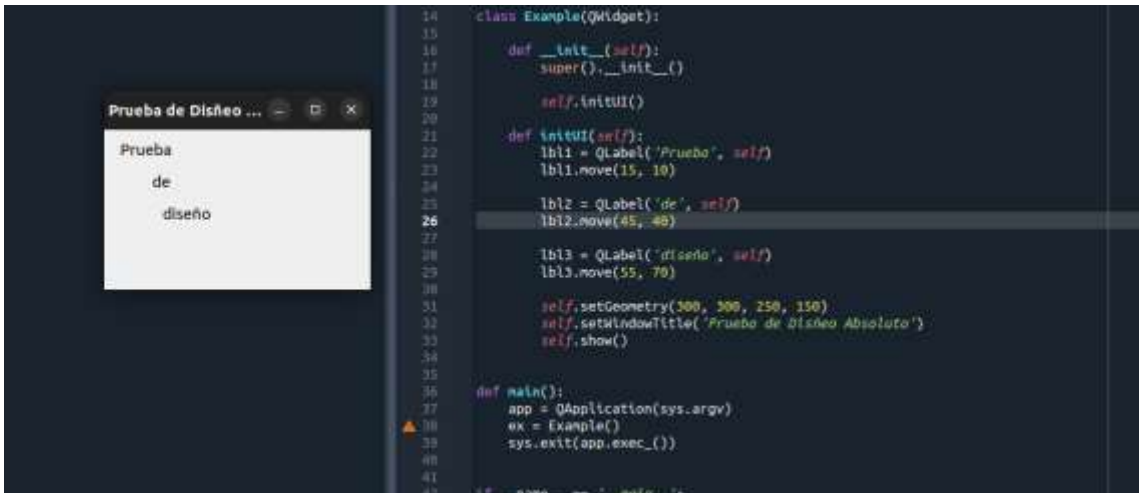

Imagen 1.3.6. Creación de una ventana con texto en posicionamiento absoluto.

QHBoxLayout y *QVBoxLayout* son clases de diseño básicas que alinean los *widgets* horizontal y verticalmente. En ocasiones es necesario colocar dos botones en una determinada posición, en este caso, la esquina inferior derecha. Para crear tal diseño, creamos dos pulsadores con *QPushButton*, a los que se le pueden modificar el nombre, usamos un cuadro horizontal y uno vertical. Para crear el espacio necesario, agregamos un factor de estiramiento y colocamos dos botones en la esquina inferior derecha de la ventana. Estos botones se mantienen en dicha esquina cuando cambiamos el tamaño de la ventana de la aplicación.

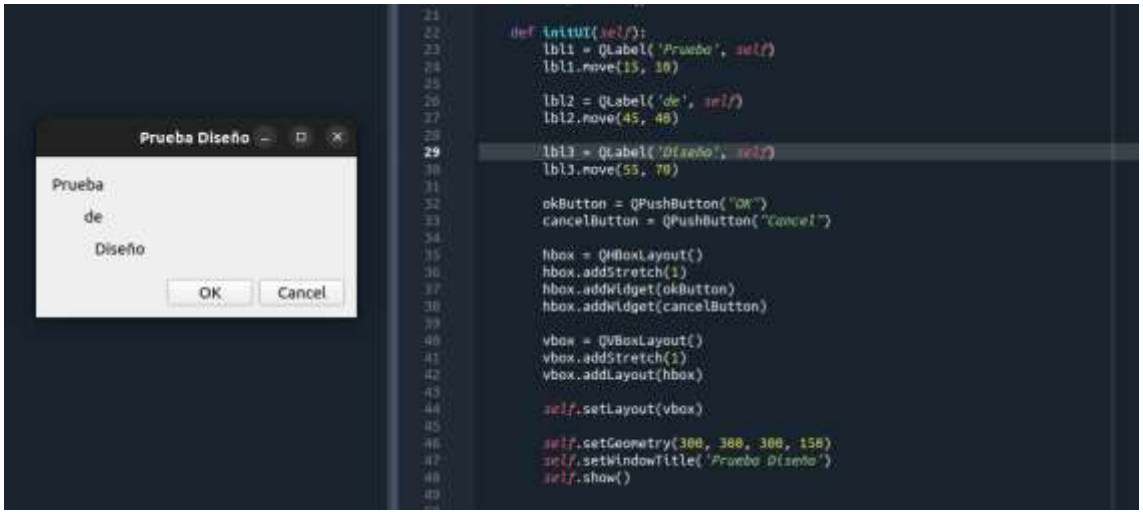

Imagen 1.3.7. Creación de una ventana con dos botones con posicionamiento en la izquierda inferior de la pantalla.

No solo podremos diseño esto, sino que además podremos dividir el espacio en filas y columnas, como, por ejemplo, realizar la cuadricula de una calculadora por medio del comando *setLayout*. La instancia de un *QGridLayout* se crea y se configura para que sea el diseño de la ventana de la aplicación. Los botones se crean y se agregan al diseño con el método *addWidget*. Un ejemplo de esto, se encontrará en el siguiente link, http://www.uco.es/~i02matog/practica1/p1_ejemplo9_calculadora.txt.

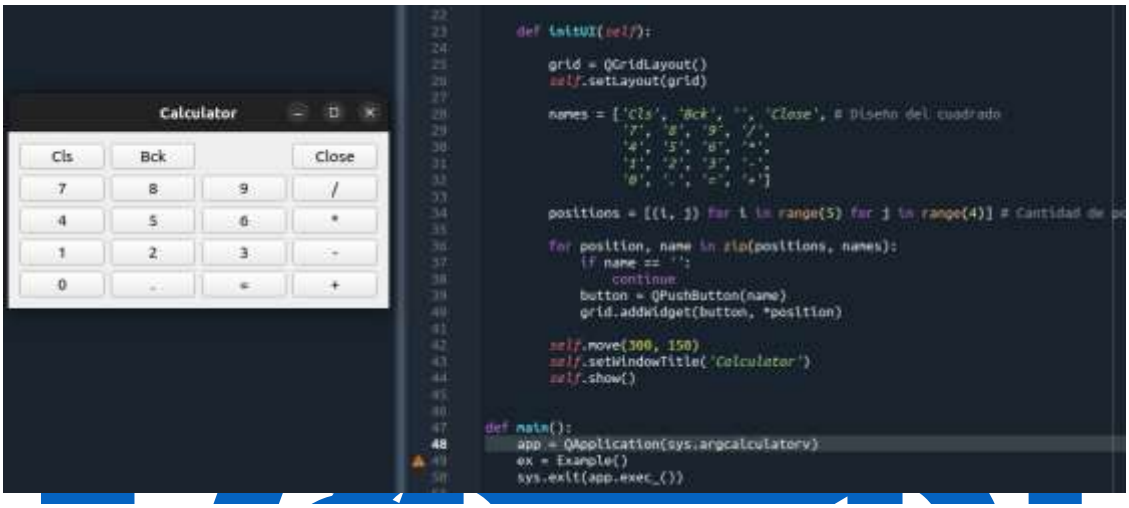

Imagen 1.3.8. Creación de una cuadrícula similar al de una calculadora.

Tam es
de
Crea
agre
rang
cinc er

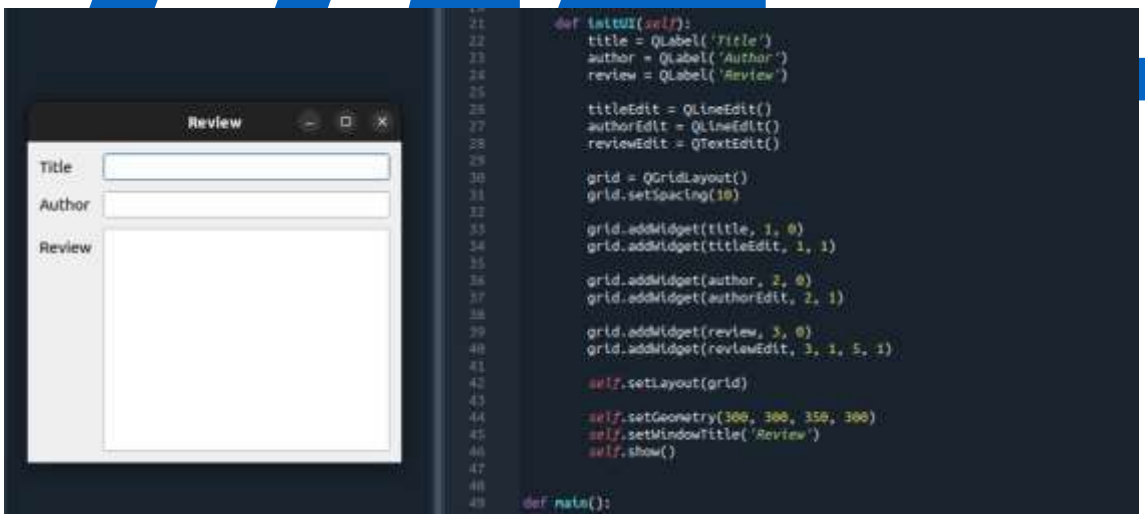

Imagen 1.3.9. Creación de una ventana a modo de creación de rewiews.

3.3. Diálogos.

En una aplicación informática, un cuadro de diálogo (conversación entre dos o más personas) es una ventana que se utiliza para comunicarse con la aplicación. Los diálogos se utilizan para acciones como obtener datos de los usuarios o cambiar la configuración de la aplicación. QInputDialog proporciona un diálogo de conveniencia simple para obtener un valor único del usuario, cuyo valor de entrada puede ser una cadena, un número o un elemento de una lista.

El siguiente ejemplo tiene un botón y un widget de edición de línea. Al pulsar el botón, se muestra el cuadro de diálogo de entrada para obtener valores de texto. El texto ingresado se mostrará en el widget de edición de línea, como se puede ver en el siguiente enlace, http://www.uco.es/~i02matog/practica1/p1_ejemplo14_dialogos.txt.

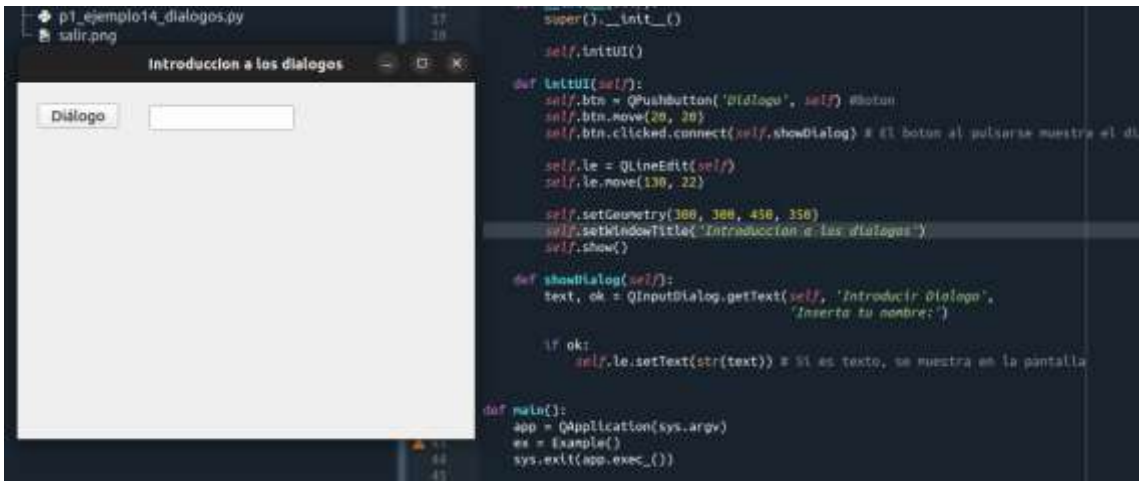

Imagen 1.3.10. Iniciación a los diálogos.

Esta línea muestra el cuadro de diálogo de entrada. La primera cadena es un título de diálogo y la segunda es un mensaje dentro del diálogo. El cuadro de diálogo devuelve el texto introducido y un valor booleano. Si hacemos clic en el botón Ok, el valor booleano es verdadero, modificando y llevando dicho valor a la ventana anterior. El código se encontrará en el siguiente link, http://www.uco.es/~i02matog/practica1/p1_ejemplo15_dialogos_colores.txt.

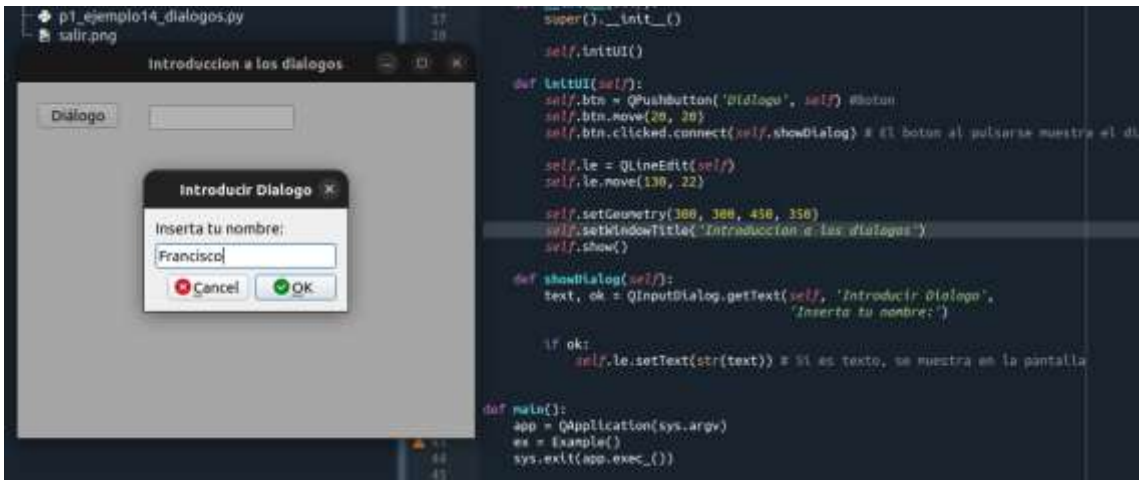

Imagen 1.3.11. Iniciación a los diálogos II.

QColorDialog proporciona un widget de diálogo para seleccionar valores de color. El ejemplo de aplicación muestra un botón pulsador y un *QFrame*. El fondo del widget está configurado en color negro. Usando *QColorDialog*, podemos cambiar el fondo, por medio de *QColor*. Si hacemos clic en el botón Cancelar, no se devuelve ningún color válido. Si el color es válido, cambiamos el color de fondo usando hojas de estilo.

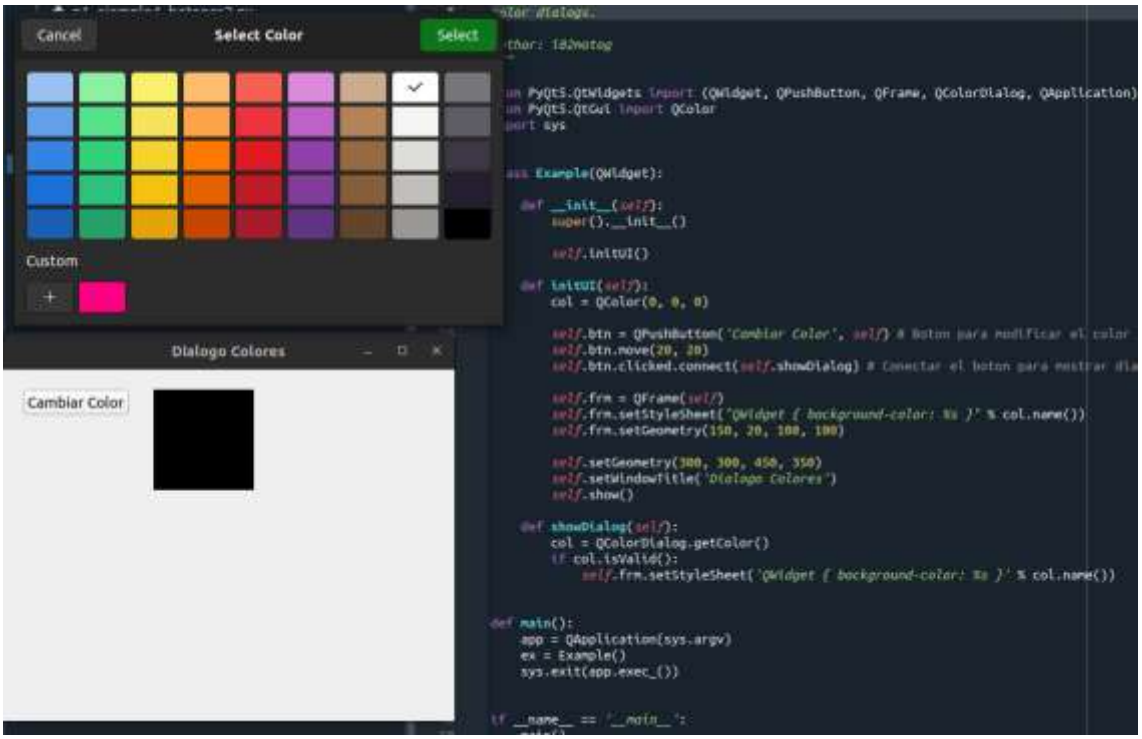

Imagen 1.3.12. Cambiar el color seleccionando otro.

No solo podemos modificar el color, sino que también podremos modificar otros aspectos como la tipografía. Con *QFontDialog*, cambiamos la fuente de la etiqueta. El método *getFont* devuelve el nombre de la fuente. Si hacemos clic en Aceptar, la fuente de la etiqueta se cambia por medio de la función *setFont,* http://www.uco.es/~i02matog/practica1/p1_ejemplo16_dialogos_fuentes.txt.

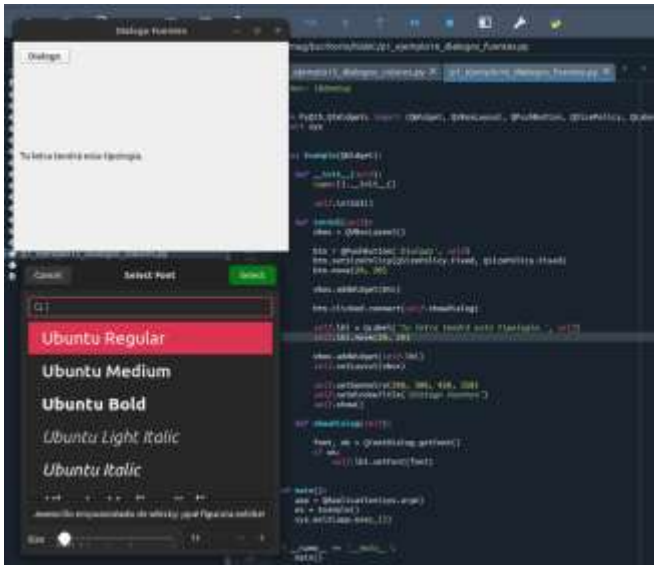

Imagen 1.3.13. Modificación de la tipología del texto.

QFileDialog es un cuadro de diálogo que permite a los usuarios seleccionar archivos o directorios. Los archivos se pueden seleccionar tanto para abrirlos como para guardarlos. El ejemplo muestra una barra de menú, un widget de edición de texto centralizado y una barra de estado. El elemento del menú muestra el *QFileDialog* que se utiliza para seleccionar un archivo. El contenido del archivo se carga en el widget de edición de texto.

El ejemplo se basa en el widget *QMainWindow* porque configuramos de forma centralizada un widget de edición de texto. La primera cadena del método *getOpenFileName* es el título. La segunda cadena especifica el directorio de trabajo del diálogo con el que trabajaremos. Usamos el módulo de ruta para determinar el directorio de inicio del usuario. De forma predeterminada, el filtro de archivos se establece en todos los archivos (*). El nombre del archivo seleccionado se lee y el contenido del archivo se establece en el widget de edición de texto. El código del programa se encontrará aquí, http://www.uco.es/~i02matog/practica1/p1_ejemplo17_dialogos_fichero.txt.

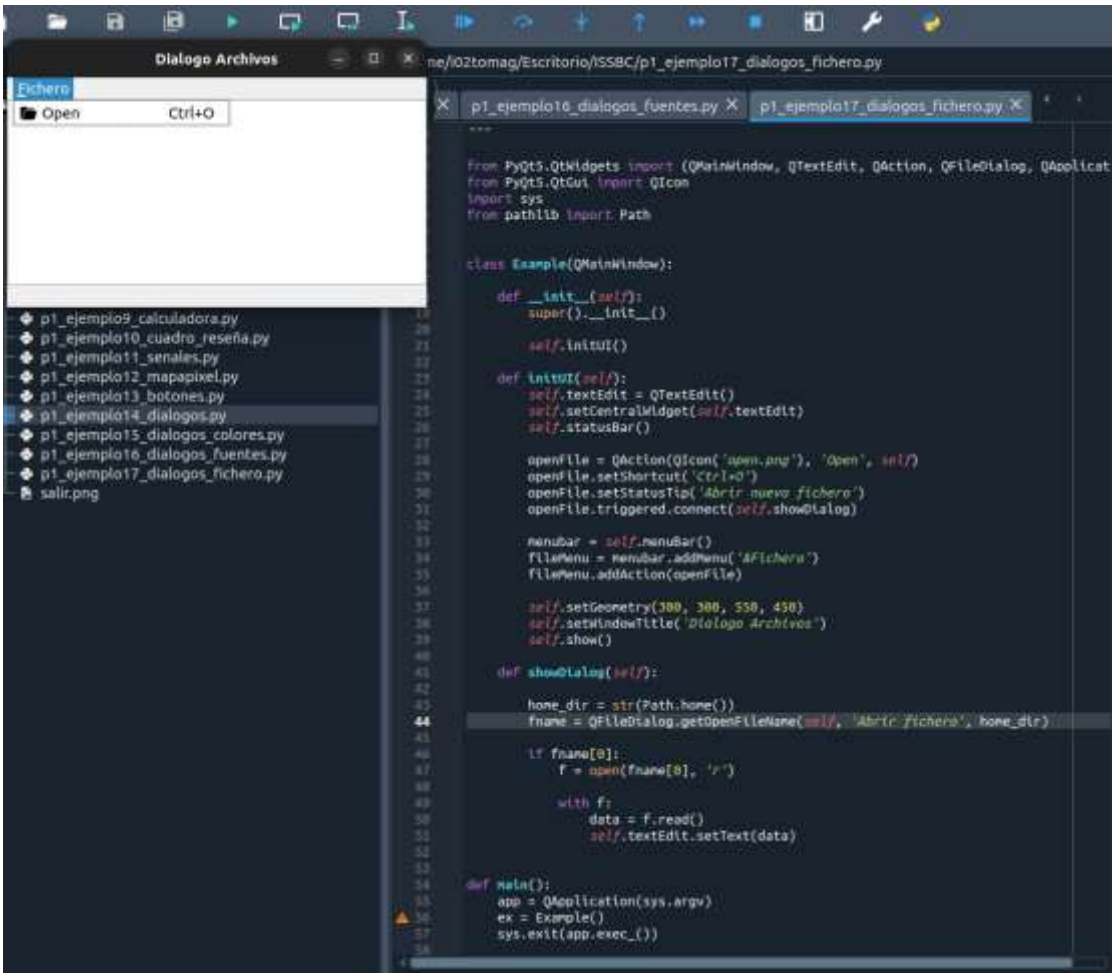

Imagen 1.3.14. Apertura de documentos mediante Widget I.

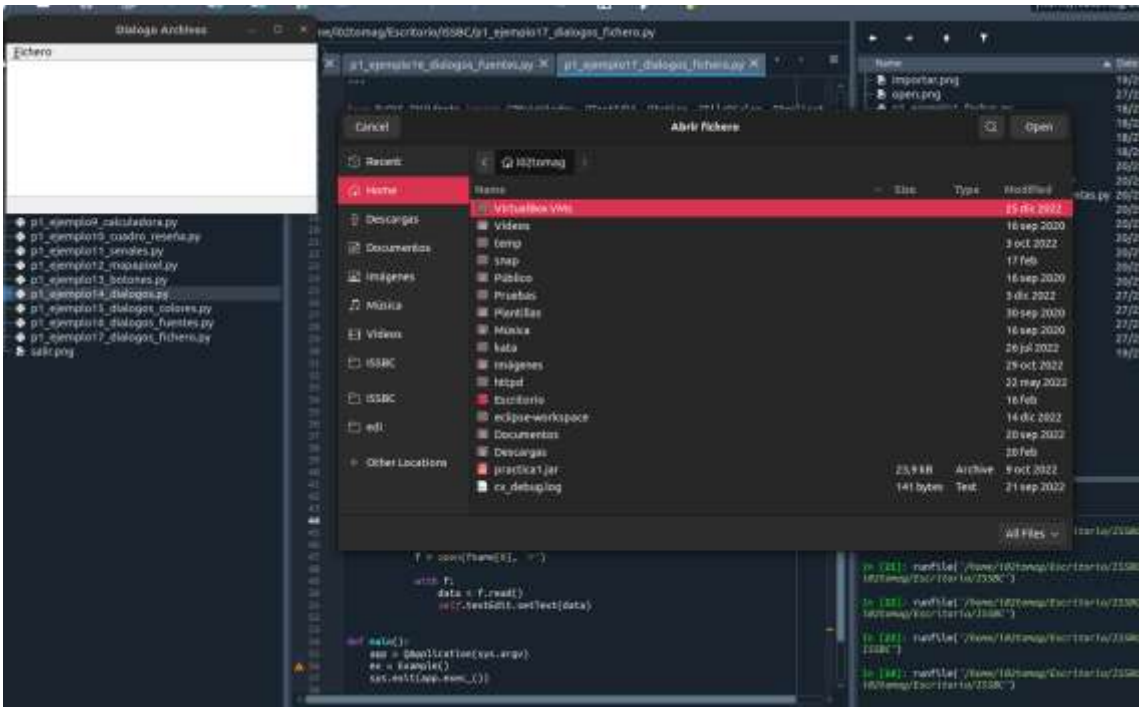

Imagen 1.3.15. Apertura de documentos mediante Widget II.

3.4. Widgets.

Los widgets son componentes básicos de una aplicación. *PyQt5* tiene una amplia gama de varios widgets, incluidos botones, casillas de verificación, controles deslizantes o cuadros de lista.

3.4.1. CheckBox.

Un *QCheckBox* es un widget que tiene dos estados, encendido y apagado, como si fuera una caja con una etiqueta, que puede variar. Las casillas de verificación se utilizan normalmente para representar funciones en una aplicación que se pueden habilitar o deshabilitar. En nuestro ejemplo, crearemos una casilla de verificación que alternará el título de la ventana. El constructor del *Checkbox* será *QCheckBox*. Con *stateChanged.connect(self.changeTitle*), el texto del título se modificará. Si el widget está marcado, establecemos un título de la ventana predeterminado anteriormente. De lo contrario, establecemos una cadena vacía en la barra de título, procediendo a la ocultación de este. Un ejemplo de este programa aparecerá en el siguiente link, http://www.uco.es/~i02matog/practica1/p1_ejemplo18_widget_titulo.txt.

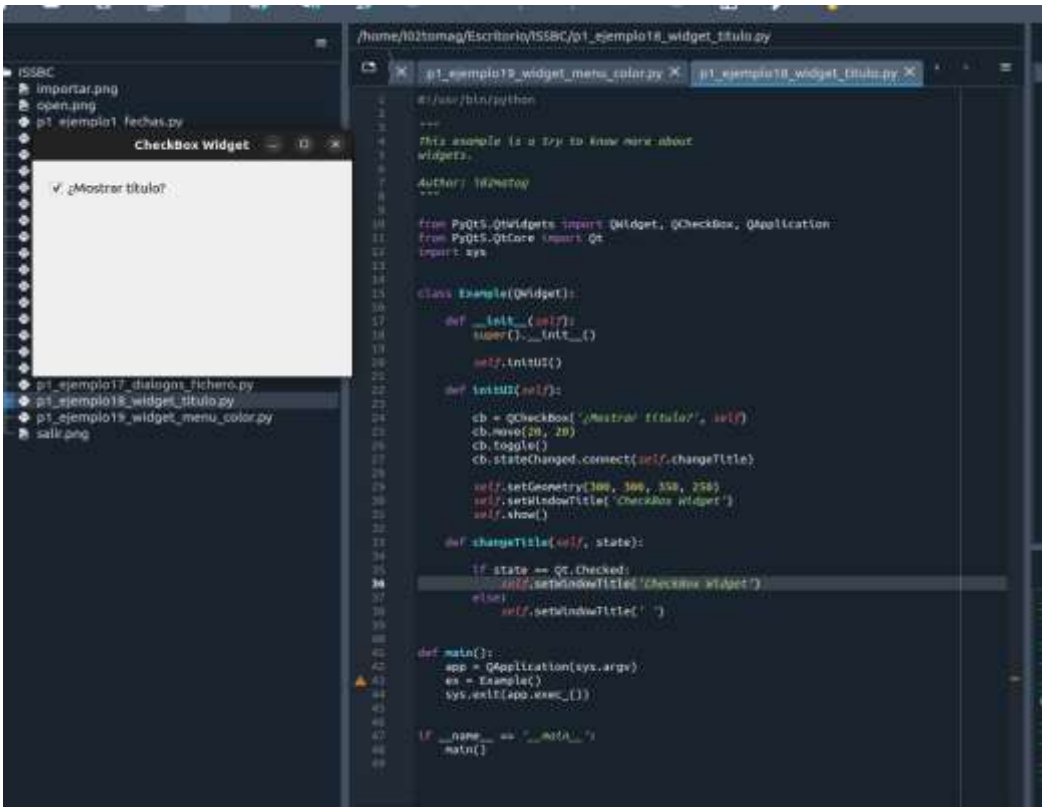

Imagen 1.3.16. CheckBox para mostrar/ocultar el título I.

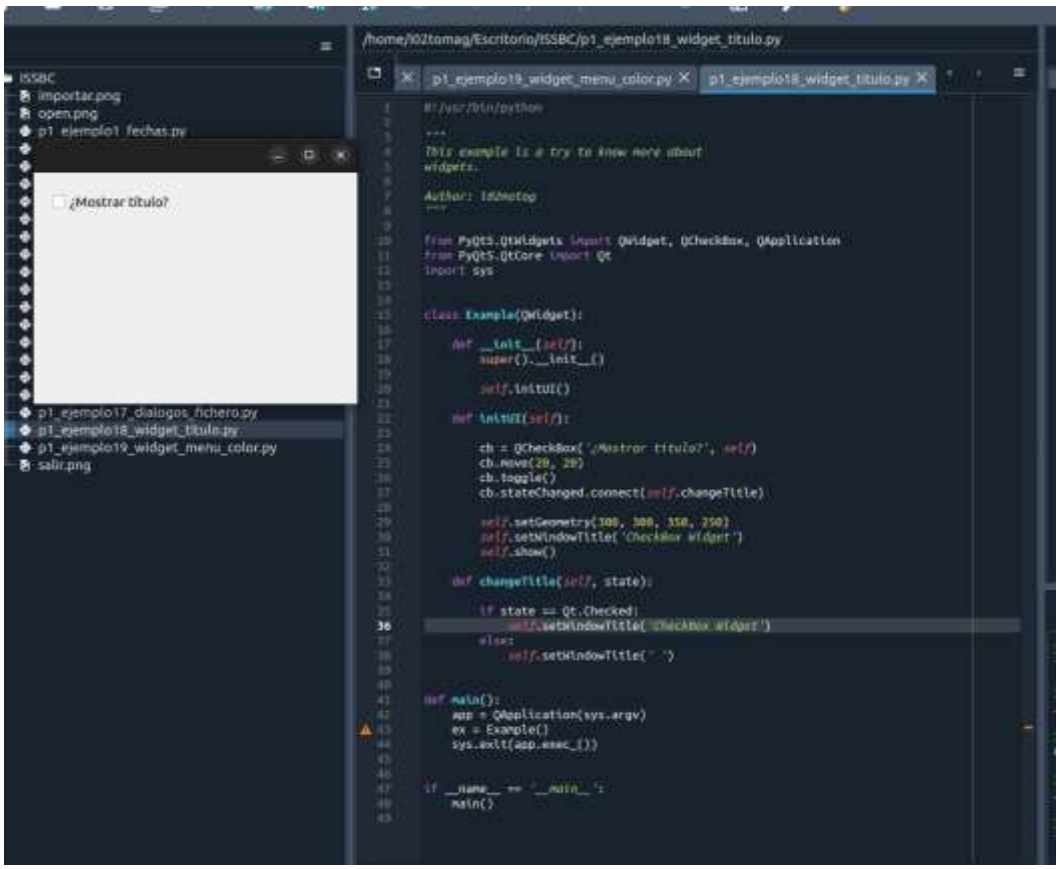

Imagen 1.3.17. CheckBox para mostrar/ocultar el título II.

3.4.2. Menú de Colores.

Incorporamos un botón de alternancia es un *QPushButton* en un modo especial, con dos estados, presionado y no presionado. Alternamos entre estos dos estados haciendo clic en él. En nuestro ejemplo, creamos tres botones de alternar y un *QWidget*. Establecemos el color de fondo del *QWidget* en negro. Los botones de alternancia alternarán las partes roja, verde y azul del valor del color. El color de fondo depende de qué botones se presionen.

Para crear un botón de alternar, creamos un *QPushButton* y lo hacemos verificable llamando al método *setCheckable*. Posteriormente, conectamos una señal de clic a nuestro método definido por el usuario y usamos la señal de clic que opera con un valor booleano. En caso de que sea un botón rojo, actualizamos la parte roja del color en consecuencia. Usamos hojas de estilo para cambiar el color de fondo, la cual se actualiza con el método *setStyleSheet*. Se podrá realizar este ejemplo, con el código de aquí, http://www.uco.es/~i02matog/practica1/p1_ejemplo19_widget_menu_color.txt.

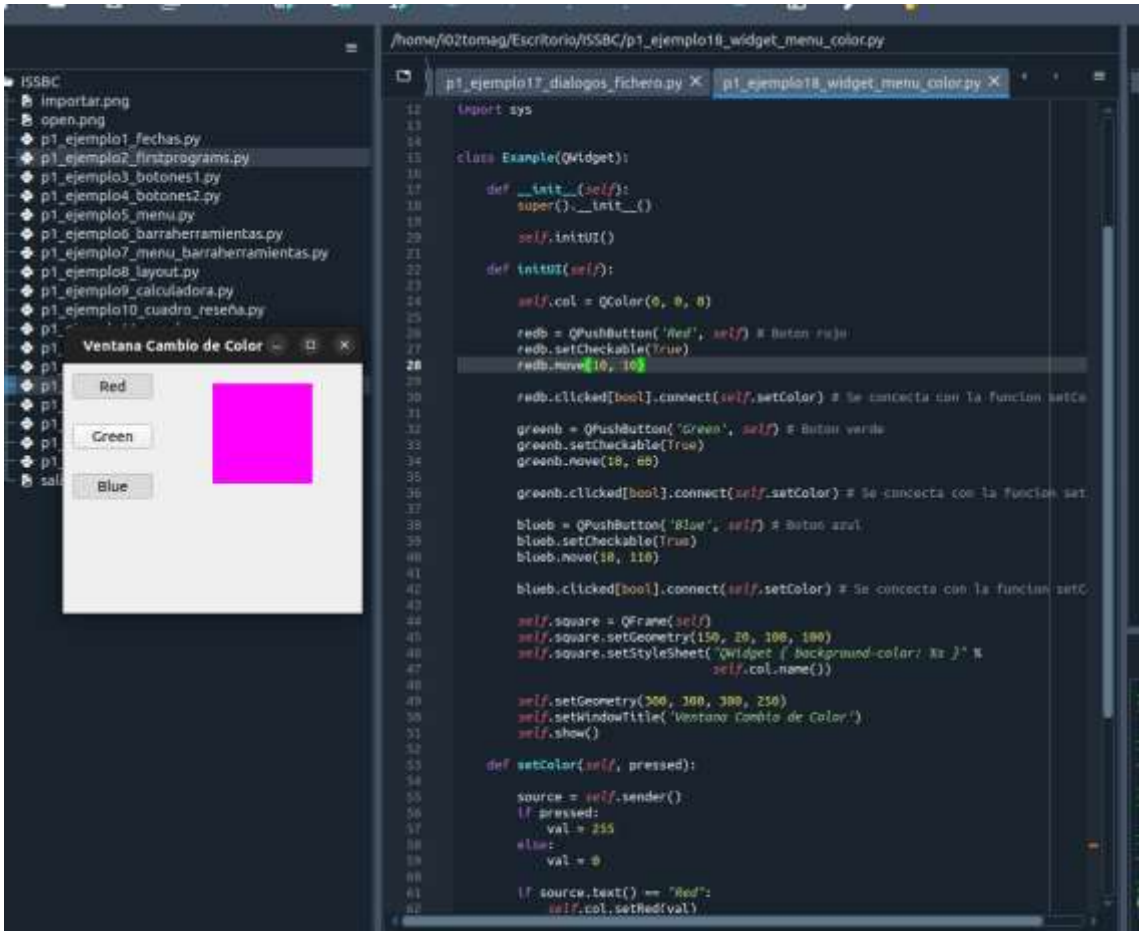

Imagen 1.3.18. Widget para modificar los colores por medio de los botones.

3.4.3. Barra de Sonido.

Un *QSlider* es un widget que tiene un manejo simple, pudiendo tanto avanzar como retroceder. De esta manera estamos eligiendo un valor para una tarea específica. A veces, usar un control deslizante es más natural que ingresar un número o usar un cuadro de número. En nuestro ejemplo, mostramos un control deslizante y una etiqueta.

La etiqueta muestra una imagen y el control deslizante controla la etiqueta. En nuestro ejemplo, simulamos un control de volumen. Al arrastrar el controlador de un control deslizante, cambiamos una imagen en la etiqueta.

Con *QSlider* creamos un *QSlider* horizontal y según el valor del control deslizante, establecemos una imagen en la etiqueta. En el código anterior, configuramos una imagen *mute* en la etiqueta si el control deslizante es igual a cero. Pudiendo introducir otros iconos en diferentes puntos en función de los rangos que introduzcamos. El código será accesible por medio del siguiente enlace, http://www.uco.es/~i02matog/practica1/p1_ejemplo20_widget_barrasonido.txt.

3.4.4. Barra de Progreso.

Una barra de progreso es un widget que se usa cuando procesamos tareas largas y se encuentra animado para que el usuario sepa que la tarea está progresando. El *QProgressBarwidget* proporciona una barra de progreso horizontal o vertical en el kit de herramientas de PyQt5. El programador puede establecer el valor mínimo y máximo para la barra de progreso. Los valores predeterminados son 0 y 99.

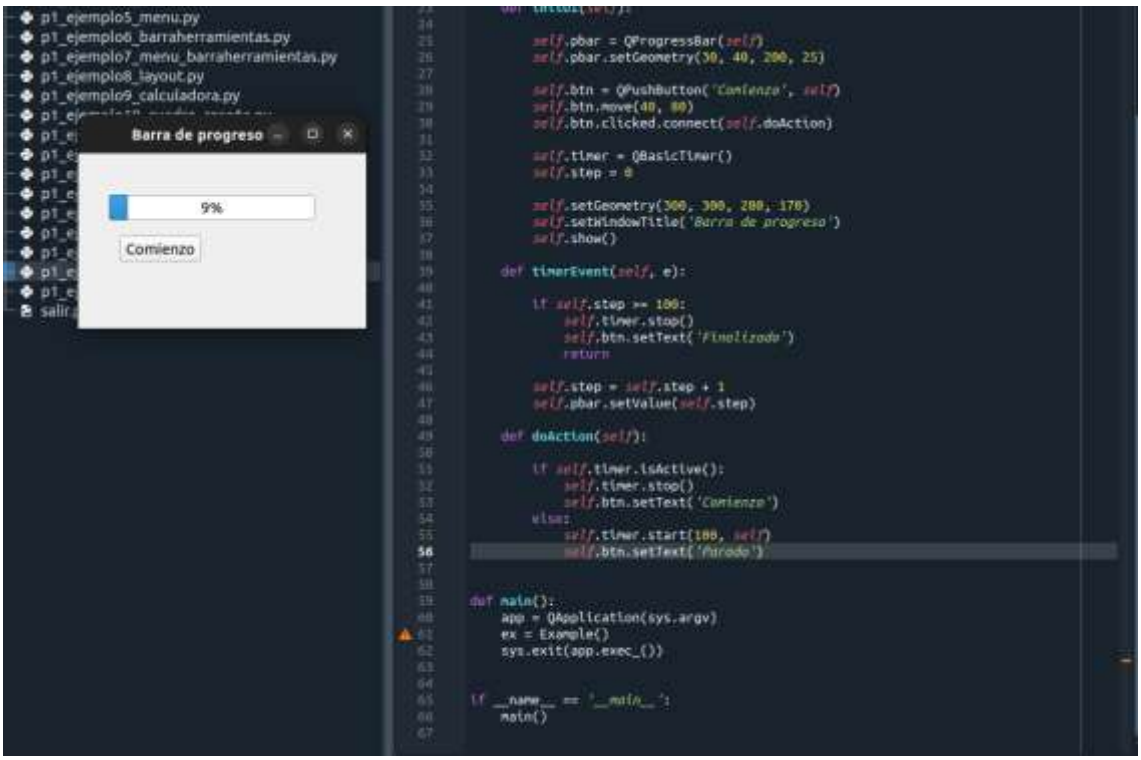

Imagen 1.3.19. Widget para introducir una barra de proceso I.

En nuestro ejemplo, tenemos una barra de progreso horizontal y un botón pulsador, donde el pulsador inicia y detiene la barra de progreso. Para activar la barra de progreso, usamos un objeto de temporizador por medio de *self.timer = QBasicTimer().* Este método tiene dos parámetros, que son el tiempo de espera y el objeto que recibirá los eventos. Cada uno *QObjecty* sus descendientes tienen un *timerEvent* (*temporizador*) controlador de eventos. Dentro del *doAction*, iniciamos y detenemos el temporizador.

Un ejemplo de este, se encontrará en el siguiente link, http://www.uco.es/~i02matog/practica1/p1_ejemplo21_widget_barraproceso.txt.

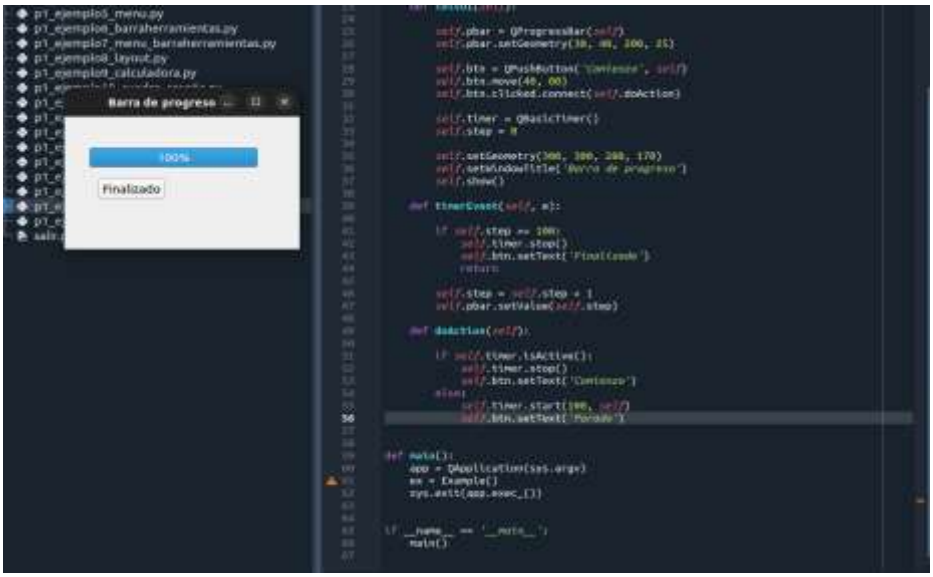

Imagen 1.3.20. Widget para introducir una barra de proceso II.

3.4.5. Calendario.

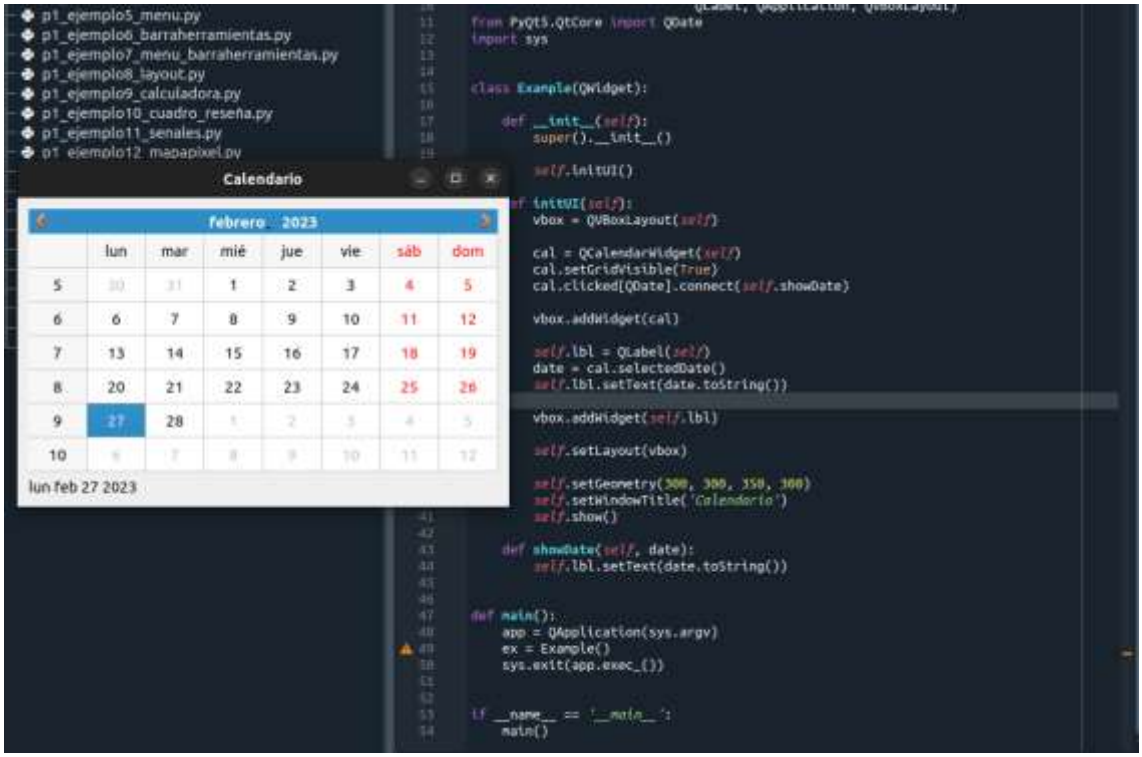

Imagen 1.3.22. Widget para mostrar un calendario.

A *QCalendarWidget* proporciona un widget de calendario mensual. Permite al usuario seleccionar una fecha de forma sencilla e intuitiva. El ejemplo tiene un widget de calendario y un widget de etiqueta. La fecha actualmente seleccionada se muestra en el widget de etiqueta. Con *QCalendarWidget* se crea el calendario. Si seleccionamos una fecha del widget, *clicked[QDate]* se emite una señal. Conectamos esta señal al

showDateMethod definido por el usuario. Luego, transformamos el objeto de fecha en una cadena y lo configuramos en el widget de etiqueta. Un ejemplo de calendario se encontrará en el siguiente enlace, http://www.uco.es/~i02matog/practica1/p1_ejemplo22_widget_calendario.txt.