Instituto Tecnológico de Buenos Aires

Guía de PyQt5

Kammann, Lucas Agustín - lkammann@itba.edu.ar
Trozzo, Rafael Nicolás - rtrozzo@itba.edu.ar

30 de julio de 2020

Índice

Comentarios: ¿Qué es esto?

En la guía hay detalles sobre instalación y manejo de comandos, así como descripciones paso a paso y aclaraciones para comprender las cosas un poco más y no cometer errores al momento de usar PyQt5. Si sufre ansiedad de código, probablemente no debería empezar por aquí, sino por los ejemplos.
Se agradece toda colaboración o contribución para con el Repositorio de GitHub, siguiendo el normal flujo de trabajo de Git con Fork y Pull-Request.

Requisitos: ¿Qué necesito para empezar?

- **Python 3:** Hay que instalar Python... fijate acá.
- **PyCharm:** Hay otras opciones como VSCode, Atom, SublimeText, Eclipse, Notepad++, Bloc de notas, del color que quieras... Se recomienda PyCharm, fijate acá.

Nota importante: según el proceso de instalación suele hacerse de forma automática o no, pero puede ser que necesites usar comandos en la consola de Windows ejecutando scripts de Python que son muy usados, como pip principalmente, o el propio intérprete de Python. Si no te funciona, es porque no lo tenés configurado en las variables de entorno del sistema, fijate cómo ¿No te funcionan?.

Figura 1: Error en la ejecución desde consola con PIP

Instalación: ¿Qué tengo que instalar?

- **PyQt5:** Es necesario instalar este paquete. Básicamente es una adaptación del framework QtCreator original empleado en C++ para Python, puede ser que la documentación en este último lenguaje este un poco incompleta. Instalamos PyQt5 ejecutando en consola de comandos de Windows, o en la consola de Linux, el comando:

  pip install PyQt5

- **PyQt5-tools/QtDesigner:** Las herramientas de diseño QtDesginer, entre otras, para usar con este framework hay que instalarlas con otro comando:

  pip install pyqt5-tools

Comandos para consola de windows

¿No te funcionan?

Para que Windows pueda ejecutar un script, bash o algún ejecutable pasando argumentos al programa desde la consola simplemente utilizando una palabra clave, es necesario que esté registrado en las variables de entorno.
Nota importante: si bien abajo explico cómo agregar variables de entorno, es un tema interesante, sirve para otras cosas y tiene vínculo con el mecanismo de Python para importar un paquete o módulo en forma absoluta, podrías mirarme, o a mi, o también a mi.

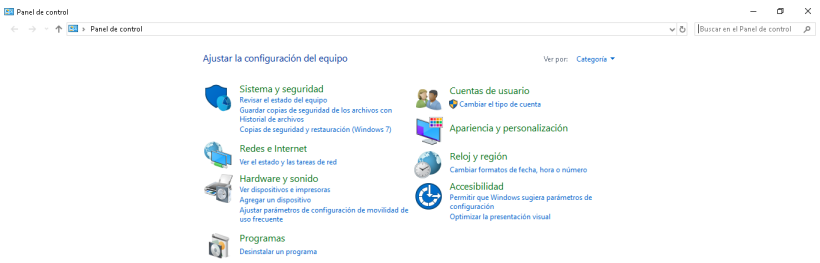


Figura 2: Abrimos Panel de Control de Windows y entramos a Sistema y Seguridad

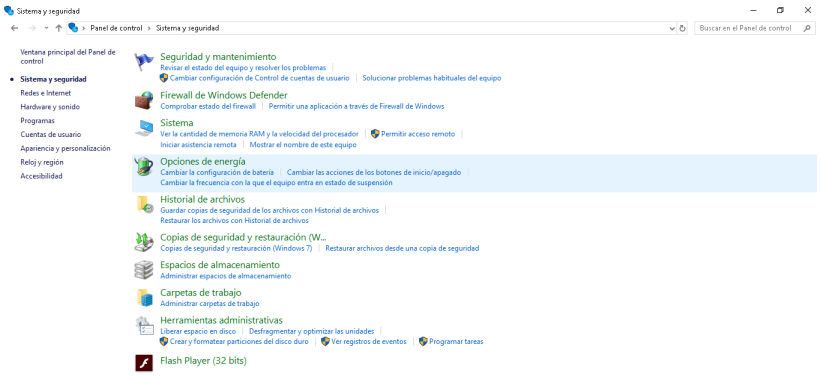


Figura 3: Entramos en Sistema

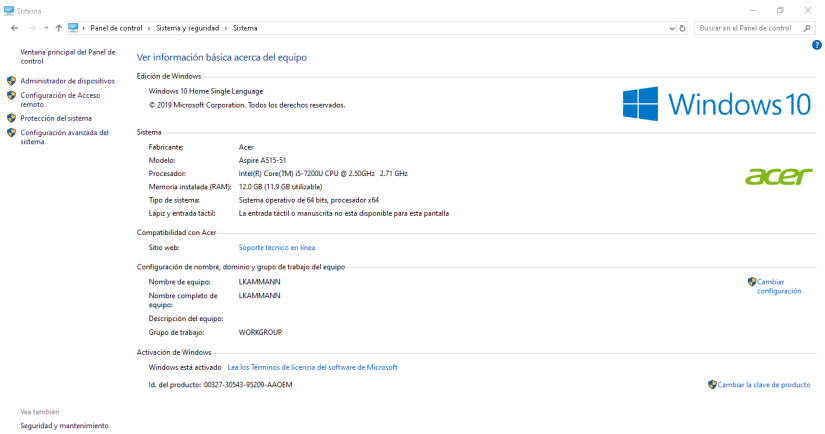


Figura 4: A la izquierda accedemos en Configuración avanzada del sistema

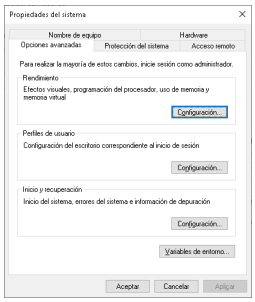


Figura 5: Entramos en el botón que dice Variables de entorno...

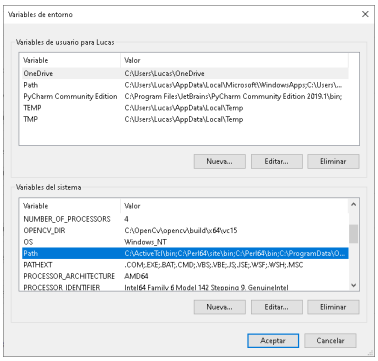


Figura 6: Buscamos en Variables del sistema la que dice Path

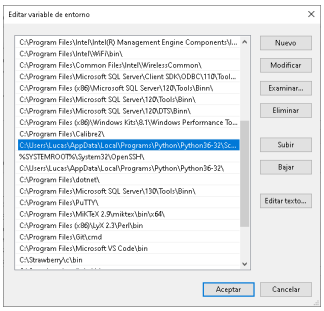


Figura 7: Accediendo a Path aparecen todo su contenido, tenemos que agregar la ubicación de Python y sus scripts

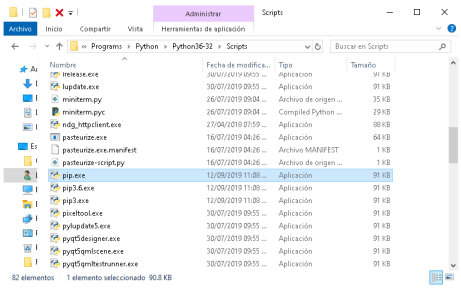


Figura 8: Ejemplo de la ubicación de los scripts de Python

Compilar los diseños de QtDesigner

Con la consola de Windows/Linux abierta, posicionado en la carpeta donde tenemos el archivo "*.ui"que creamos y editamos con el QtDesigner, luego corremos la siguiente línea para compilarlo. Sobre las ventajas o desventajas de este método para utilizar los diseños que realizamos en QtDesigner, es necesario leer otra sección del documento.

pyuic5 -x filename.ui -o output.py

Compilar los recursos de QtDesigner

Con la consola de Windows/Linux abierta, posicionado en la carpeta donde tenemos el archivo "*.qrc"que creamos y editamos con el QtDesigner, luego corremos la siguiente línea para compilarlo.

pyrcc5 filename.qrc -o output.py

Herramienta kt

¿Qué es?

Supongamos que queremos seguir la estructura de un proyecto de PyQt5 sugerida por esta guía, ¿Cómo estructurar los proyectos?, porque resulta práctico para el manejo de muchos archivos y cuando nos acostumbramos es agradable visualmente. Luego, cada vez que agregamos un Widget, al haberlo creado en QtDesigner debemos guardarlos en la carpeta designer, y además compilarlo junto con sus recursos. Y finalmente guardar los resultados de la compilación en sus respectivas carpetas. También tenemos que repetir el proceso cada vez que actualizamos o cambiamos algo, esto puede ser un poco tedioso.
Para resolver estos inconvenientes, subí un script para automatizar estos procesos, de forma muy sencilla.

Instalación

El proceso de instalación consiste en clonar el repositorio, o simplemente en algún lado de nuestro sistema guardar el script "kt.py", luego agregarlo a las variables de entorno del sistema operativo para usarlo desde cualquier lado.

- 1. Clonamos el repositorio de GitHub.
- 2. Agregamos a las variables de entorno la dirección a la carpeta /tools dentro del repositorio. En ¿No te funcionan?, se explica como agregar esa variable, pero con Python.
- 3. Listo!

Uso

Creación de un proyecto

Cuando queremos crear un proyecto, es decir, crear la estructura del directorio correspondiente a lo convenido por la guía, debemos ir a la carpeta principal del proyecto, y ejecuta la siguiente línea. El uso del nombre de la aplicación está sujeto a la versión de la herramienta, en su creación no era usado pero se dejó por futuras funcionalidades.

kt.py app

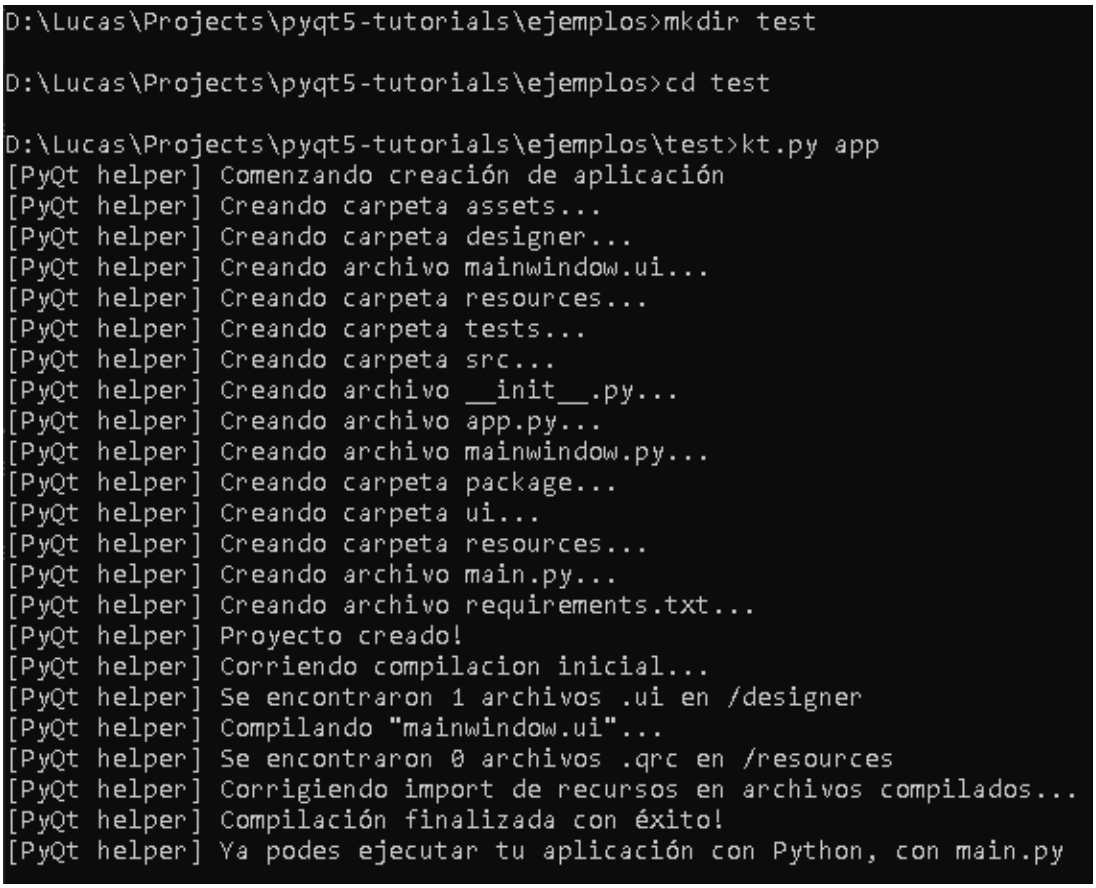


Figura 9: Creación de proyecto con herramiento kt

Compilación de .ui y .qrc

La compilación de los archivos .ui y .qrc antes era tediosa, teníamos que compilar uno por uno, luego debíamos corregir un bug del QtDesigner, que al compilar importaba mal los recursos. Y luego mover correctamente a las carpetas de la estructura convenida, pero ya no más. En la carpeta del proyecto, debemos ejecutar,

kt.py compile

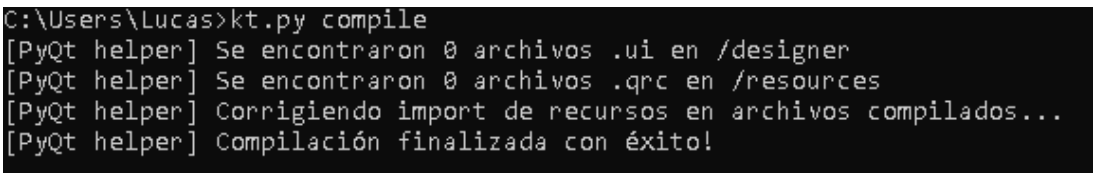


Figura 10: Compilación de proyecto con herramiento kt

Figura 11: Compilación de proyecto real con herramienta kt

Distribución .exe con .dll's

Si desean crear una distribución ejecutable sin depender de tener el interprete de Python, o proteger su código, se puede crear un archivo ejecutable en conjunto de las librerías dinámicas que fueron utilizadas. Para ello,

kt.py build

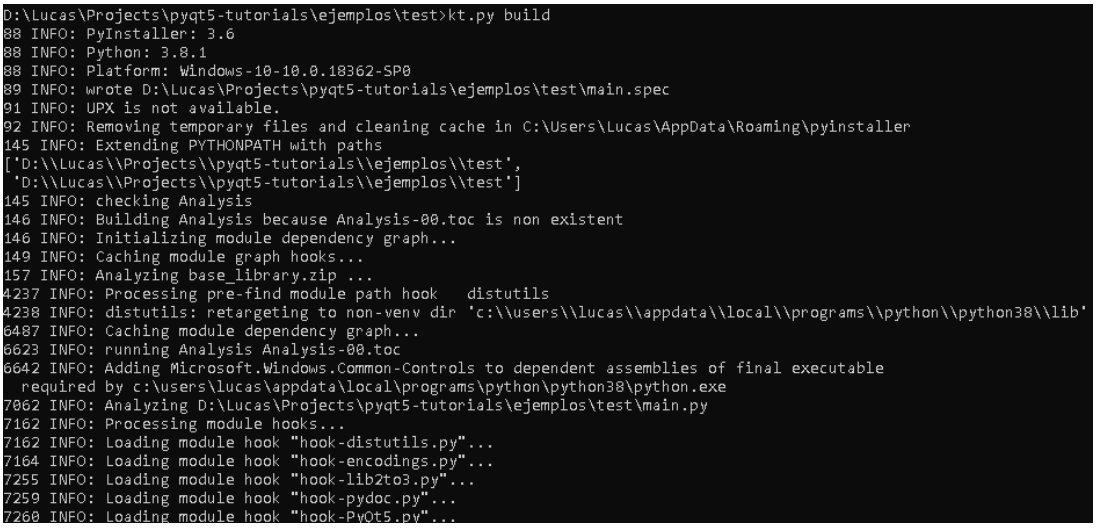


Figura 12: Creando ejecutable con kt

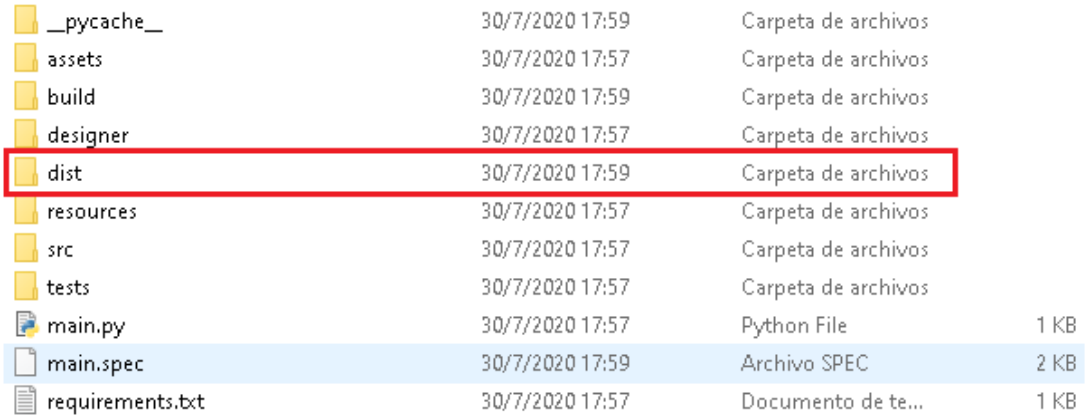


Figura 13: Carpeta distribuible

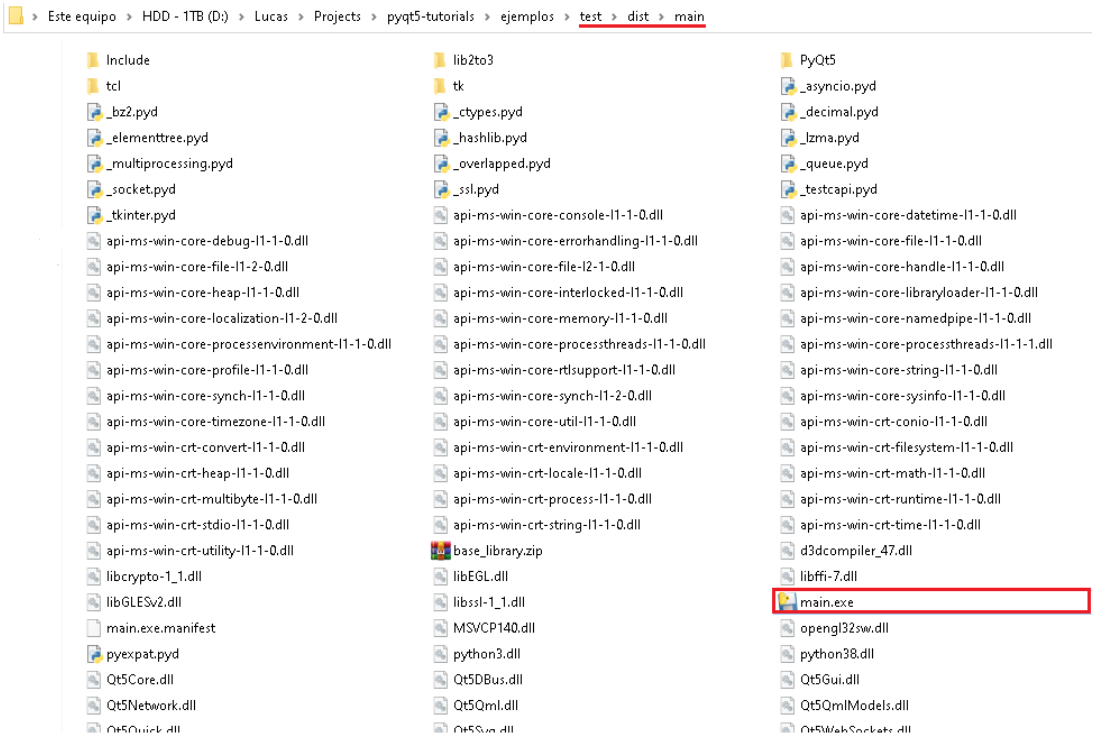


Figura 14: Archivo ejecutable en la carpeta

QtDesigner: ¿Cómo se usa?

La idea principal de esta sección es orientar en el uso de cada parte de la interfaz de QtDesigner, siguiendo el desarrollo de algunos ejemplos simples que pueden encontrarse también en la carpeta de ejemplos del Repositorio de GitHub.

Creación de un Form

Abrimos QtDesigner, o en New File..,vamos a encontrar esta ventana. Nos deja crear una nueva form de entre esas opciones. Tenemos que tener en mente que **Widget** será cualquier elemento dentro de una interfaz, **Main Window** será una ventana principal que manejará el flujo principal del programa, y **Dialog** es una ventana emergente (¡de hecho esta ventana de QtDesigner es un Dialog!).

- No deberíamos tener más de un Main Window en nuestro proyecto.
- Siempre que creemos entre estas tres opciones, nuestra clase heredará alguna de ellas. QWidget, QMainWindow o QDialog.
- Existen dialogs (QDialog) ya provistos por el framework, fijate después Ventanas de diálogo para reutilizar.

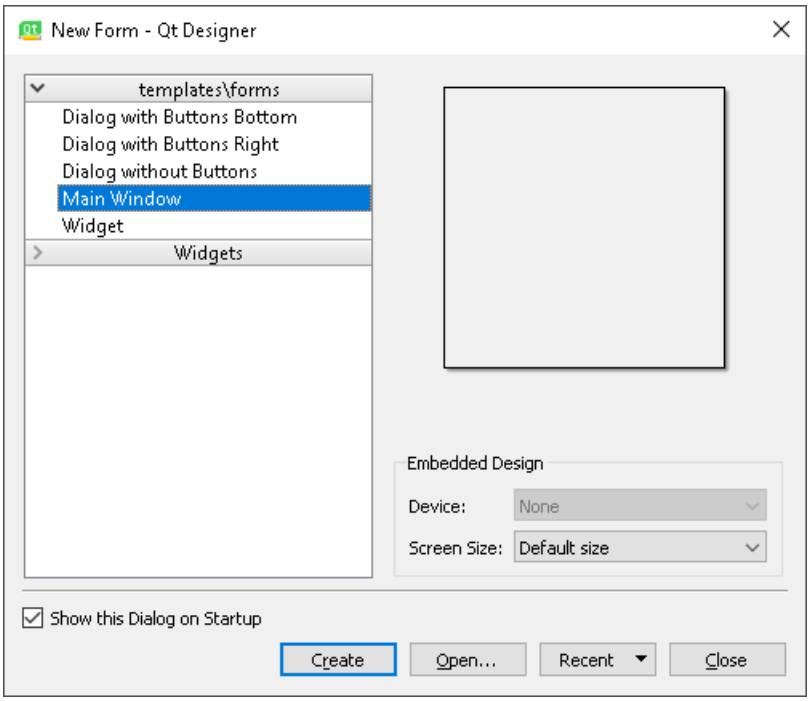


Figura 15: Ventana de creación de formas

¿Qué es cada cosa en la IDE?

Es importante conocer la organización general de la IDE que estamos usando, saber de qué es capaz puede ser un conocimiento útil para solucionar problemas fácilmente. En primer lugar veamos en la Fig. 16 algunas cosas elementales, la WidgetBox, nuestro espacio de trabajo y diseño, y las opciones de Layout del Widget.

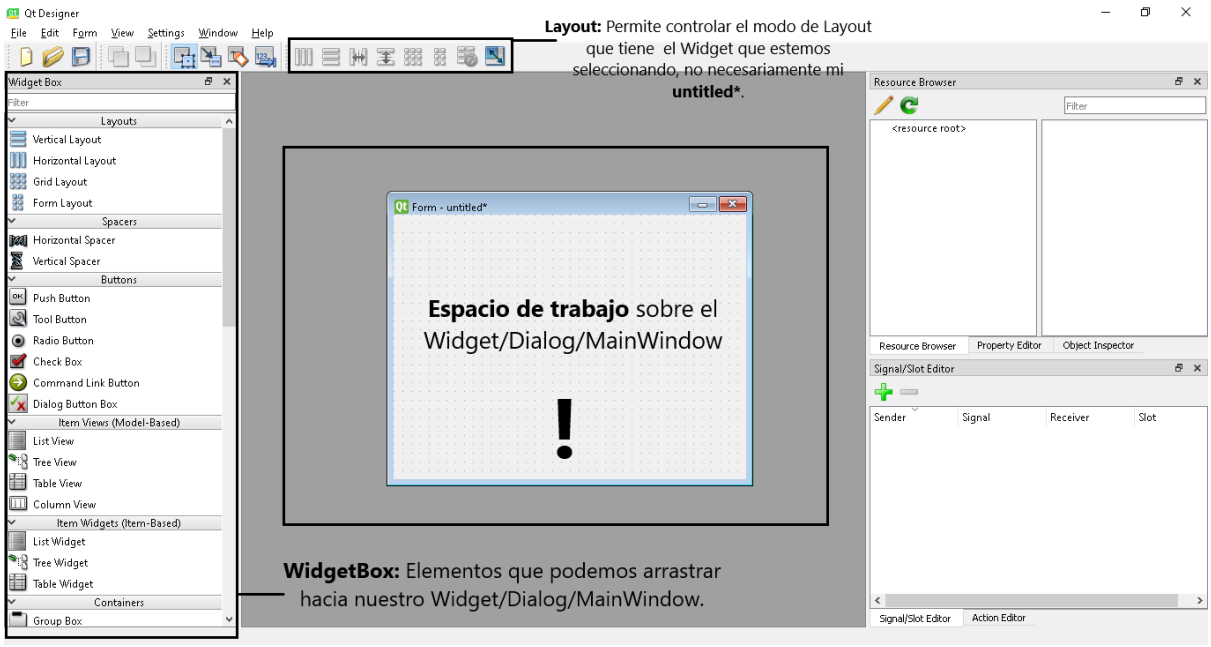


Figura 16: Esquema inicial de la IDE

Ahora, para continuar con la descripción del ejemplo vamos a hacer algunas cosas sencillas. Recordemos, no importa no entender algunas cosas de cómo es PyQt, sino reconocer las partes de la IDE, por ahora... Entonces agregamos un HorizontalSlider, un LCDNumber y luego aplicando un VerticalLayout, se obtiene esta Fig. 17.

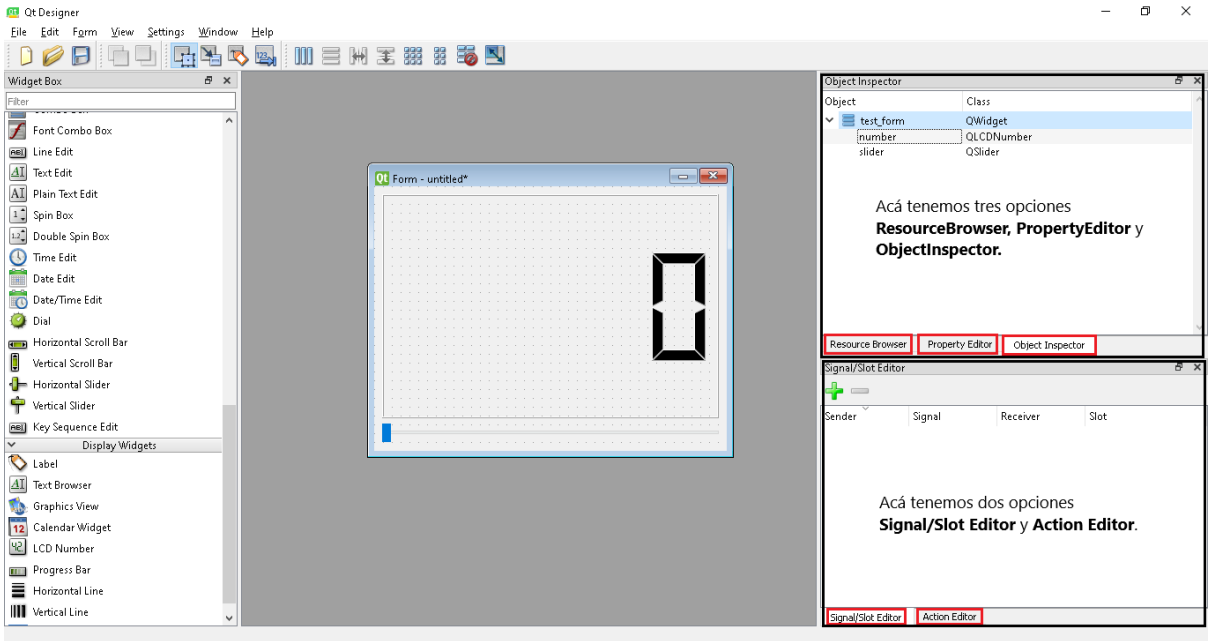


Figura 17: Editores más avanzados

Object Inspector

En PyQt existen tres elementos vamos a estar creando, que son el QWidget, QMainWindow y el QDialog. No obstante toda entidad que utilicemos hereda eventualmente de un QWidget, y armar una ventana es poner QWidgets uno dentro de otros de forma organizada para armar nuestra aplicación, lo cual nos

debería dejar la idea de que internamente esto se comporta como un árbol, y el mismo se puede visualizar en el Object Inspector. En el ejemplo vemos un QWidget, que internamente posee dos objetos y el campo **Object** es el nombre de la instancia durante ejecución, con lo cual esto debe tener un nombre adecuado a la convención de Python de forma representativa.

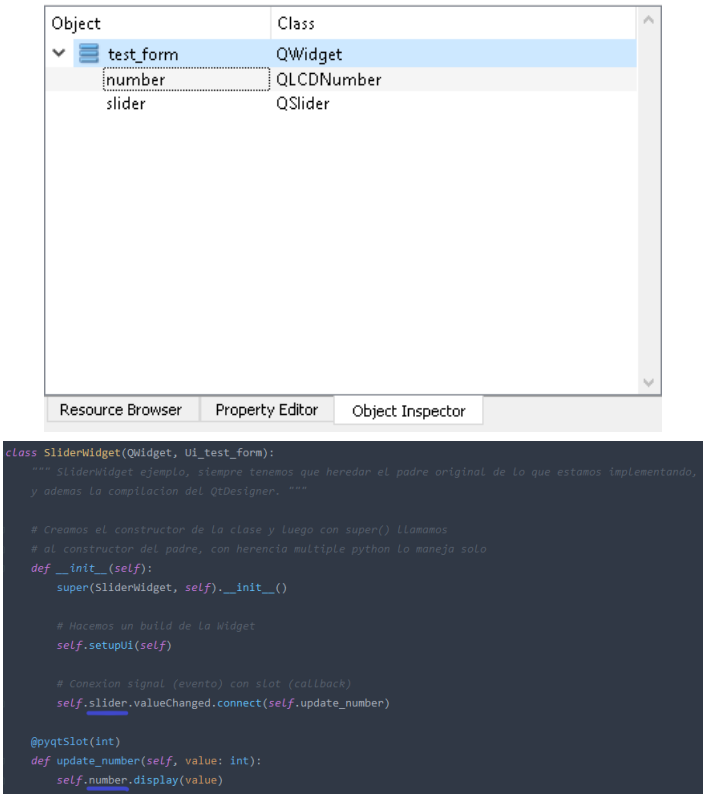


Figura 18: Object Inspector de QtDesigner

Property Editor

Cada uno de los objetos que pertenecen al árbol que muestra el Object Inspector, está construido con un conjunto de clases a través del uso de un patrón de diseño de composición. Lo importante es que podemos editar propiedades o atributos de cada una de esas partes que componen a nuestro objeto.

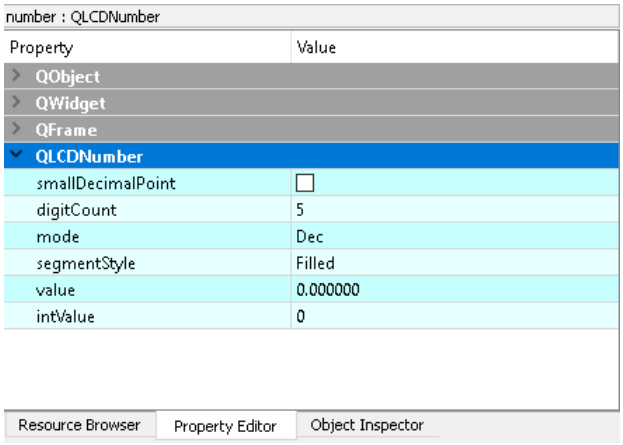
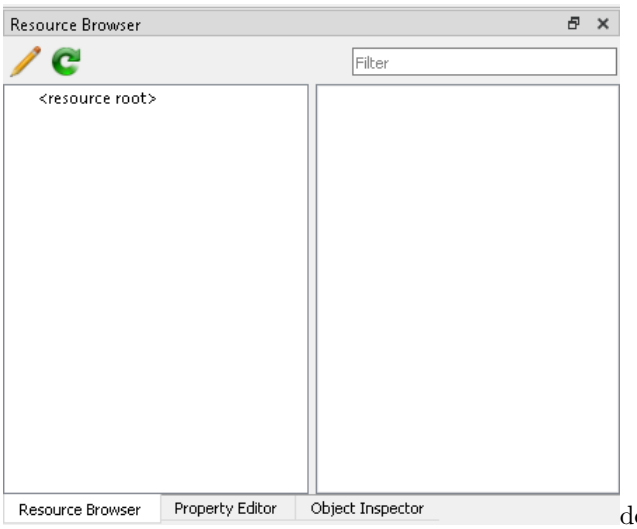


Figura 19: Property Editor de QtDesigner

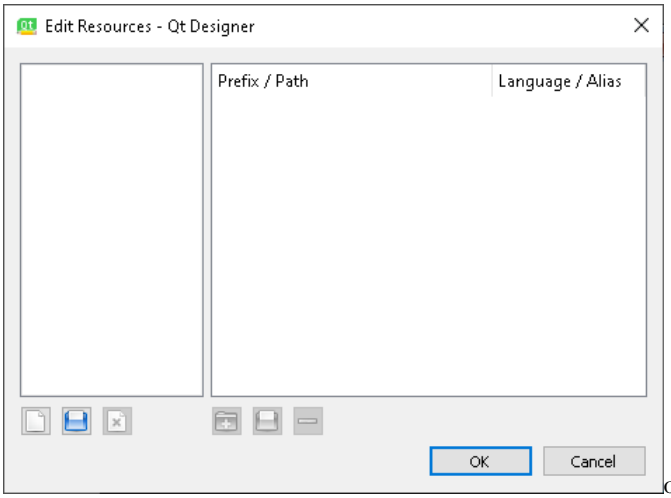
Resource Browser

El Resource Browser es muy importante, básicamente para poder crear una ventana en la cual queremos incorporar contenido personalizado, sean imágenes, audio o videos, e incluso archivos... es necesario crear lo que se define como un **Resource**. Cuando lo creamos, internamente podemos crear subcarpetas para clasificar el contenido, por ejemplo, si queremos usar algunas imagenes para un botón y otras para otro botón, de esta forma cuando queramos utilizarlas estarán disponibles.



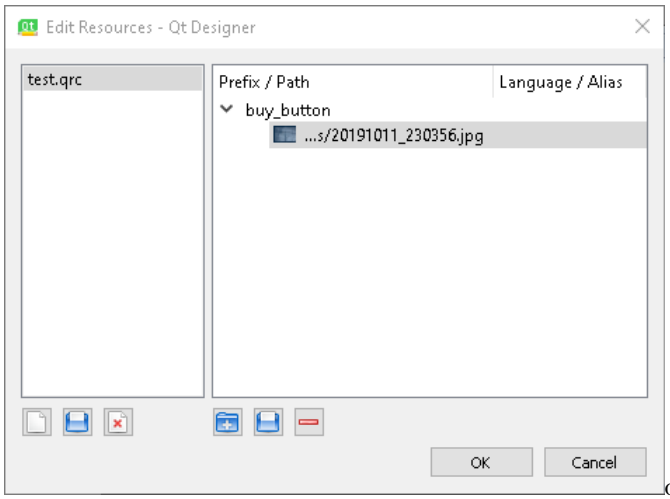
de

Figura 20: Resource Browser: Ventana de recursos, apretando el lapiz las administramos



de

Figura 21: Resource Browser: Con la hoja abajo a la izquierda creamos nuevos recursos.



de

Figura 22: Resource Browser: Agregamos Paths y sus contenidos

En primer lugar, tené en cuenta que esto genera un archivo que según el ejemplo se llama **"test.qrc"**, este mismo va a tener que ser compilado al igual que el diseño de la interfaz. Esto último será explicado más adelante.

Signal/Slot Editor

La IDE de QtDesigner tiene 4 modos de funcionamiento para el diseño de la aplicación y siempre nos encontramos en el modo de edición de Widgets. Otro modo posible es el de edición de Signal/Slots, de lo cual podes aprender qué es en ¿Qué son Signals, Slots?, mientras tanto consideremos que son eventos y callbacks para poder conectar funcionalidades de la aplicación. Este proceso de conexión se puede hacer por código siempre, pero a veces se puede hacer en el modo de edición mencionado, usando el botón de la barra de herramientas o con F4.
En la Fig. 23 se puede ver que nos aparece en rojo los Widgets, seleccionamos uno y arrastramos hacia otro, al conectarse nos permite elegir una Signal del primero con un Slot del seguno a ser ejecutado. En este caso puedo utilizar sliderMoved(int) con display(int). Es importante que hayan Signal y Slot compatibles entre los Widgets a conectar.

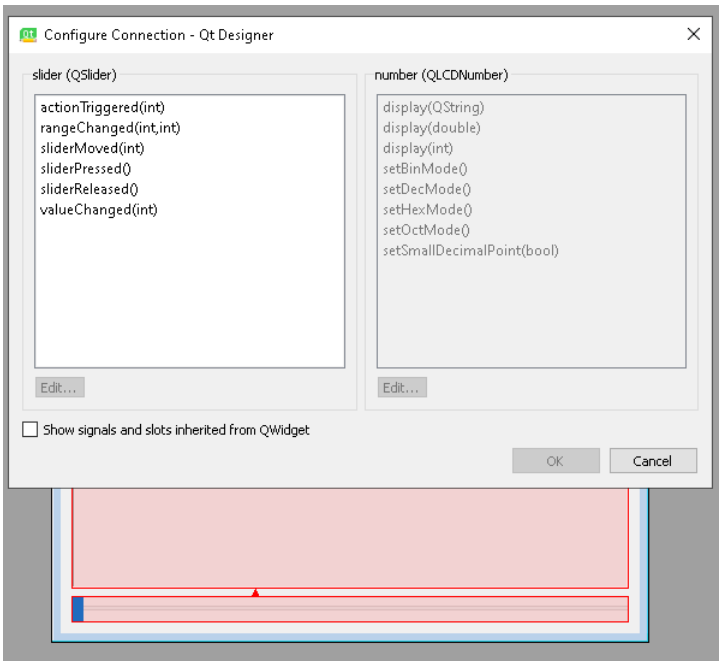


Figura 23: Modo de edición por Signal y Slot

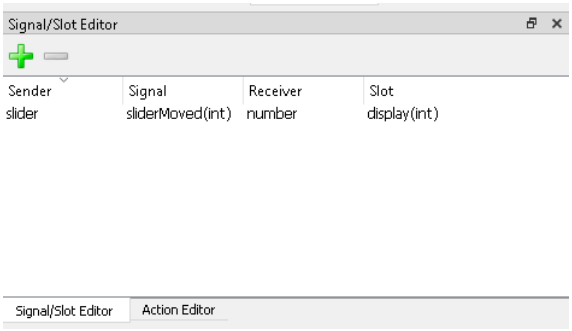


Figura 24: Ventana de conexiones de Signal y Slot

Action Editor

En el diseño de aplicaciones de escritorio se usa mucho el concepto de Comandos, proveniente del patrón de diseño, donde lo que se busca es que una acción que es comúnmente ejecutada por diversos eventos esté disponible de una forma distinta que un callback o slot. Eso es una QAction en PyQt. Para que quede claro, la opción Guardar, en una aplicación es un comando, y en PyQt se haría como un QAction. De esta forma se puede vincular para que sea disparada (.triggered()) por diferentes Widgets. Sí, es casi lo mismo que un Slot.
Estas acciones pueden ser configuradas directamente en QtDesigner para que vincular que sean ejecutadas por algún evento dentro de la aplicación desde la IDE.

Layouts y Containers: ¿Qué son?

Por el momento lo que deberíamos saber es que cuando creamos nuestro Form, sea QWidget, QDialog o QMainWindow, inicialmente vamos a poder agregarle componentes a la GUI, no obstante cuando se lo crea por primera vez tales elementos quedan posicionados en ubicaciones **flotantes**. Para garantizar que la aplicación quede bien ordenada, y que el motor de Qt sea capaz de hacer el build y el resize, es necesario utilizar Layouts y Containers.

Concepto clave: Todo elemento de la GUI es un QWidget eventualmente, y todo QWidget puede albergar otros QWidgets, esto lo convierte en un container. Esta relación jerárquica (árbol), requiere la definición de layouts para determinar cómo recalcular los tamaños y posiciones de widgets internos.

Layout: Existen cuatro tipo de Layouts, básicamente describen la forma en que se ordenan los Widgets dentro de un Widget. Puede ser **Vertical** u **Horizontal**, en tales casos se crea una pila de los Widgets en la orientación mencionada. Por otro lado está **Grid** que los ordena por filas y columnas, y finalmente **Form**, que es un Grid que siempre tiene dos columnas. El layout se configura fácilmente con la barra de herramientas o click derecho en el Widget.

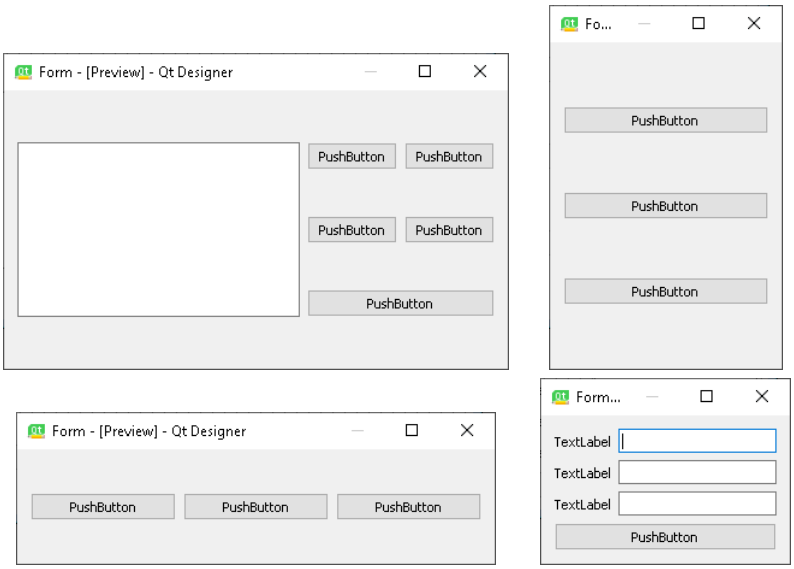


Figura 25: Ejemplos de los Layouts

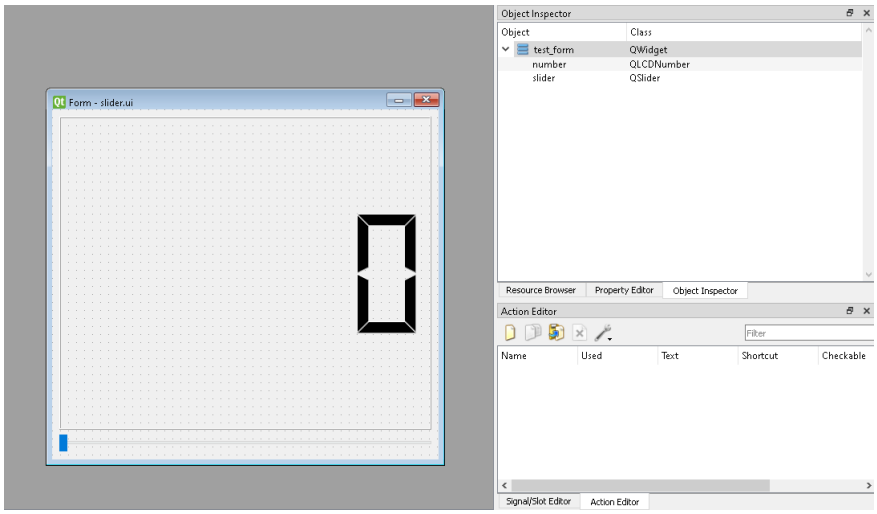
Containers: Existen muchos tipos de containers, GroupBox, ScrollArea, ToolBox, TabWidget, StackedWidget, étc. La idea principal, es que definen una forma característica de contener otros Widgets, por ejemplo, cuando utilizamos StackedWidget podemos desplazarnos entre Widgets como si fueran páginas, a voluntad del usuario, o según lo programemos en Python.

Paso a paso, ¿Cómo creo una aplicación?

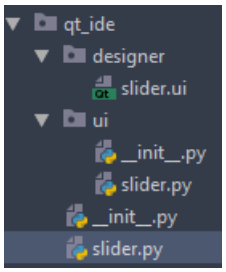
Asumimos que lo que se explica en las otras partes sobre QtDesigner ya se leyó, y esto sólo explica los pasos generales a seguir para llegar de principio a fin a nuestra aplicación ejecutable. Detalles sobre el diseño competen a otra sección.

- Creamos el Form que deseamos en QtDesigner. Sea QWidget, QDialog o QMainWindow.
- Agregamos los Widgets y definimos un Layout, organizamos nuestra GUI.
- Procuramos utilizar nombres representativos y bien definidos en el PropertyEditor. Es agradable usar las convenciones de Python, dado que luego se accederán como miembros de la clase.
- Guardamos el archivo en nuestra carpeta, al igual que los demás archivos como Resources si usamos.
- Luego realizamos el proceso de compilación ejecutando los comandos, Compilar los diseños de QtDesigner y Compilar los recursos de QtDesigner.
- A partir de esto obtenemos nuestro archivo .py compilado.

Hasta este punto, se ilustran a continuación las figuras que describen estos pasos, así como la estructura propuesta de archivos para ello, que puede ser revisada en ¿Cómo estructurar los proyectos?.




```
C:\Users\Lucas\source\repos\pyqt5-tutorials\ejemplos\qt_ide\designer>pyuic5 -x slider.ui -o slider.py
```



En la carpeta designer coloco los archivos propios del QtDesigner, luego en /ui los resultados de la compilación. Finalmente en un nivel superior de directorio, creo un nuevo archivo slider.py en el cual crearé propiamente la clase, la construiré y luego realizaré las conexiones de la GUI con el backend de mi aplicación.

```python
# PyQt5 Modules
from PyQt5.QtWidgets import QApplication
from PyQt5.QtWidgets import QWidget

# Project Modules
from ui.slider import Ui_test_form


class Slider(QWidget, Ui_test_form):
    # Creamos nuestra clase Slider, hacemos que herede del resultado de la compilacion
    # y de su Padre original en este caso QWidget.

    def __init__(self):
        # Creamos el constructor y llamamos al constructor de los padres
        # luego utilizamos el metodo para la inicializacion de los widgets internos
        super(Slider, self).__init__()
        self.setupUi(self)


if __name__ == "__main__.py":
    # Contexto o aplicacion
    app = QApplication([])
```

```
# Instancia para probar nuestra Widget
widget = Slider()
widget.show()

# Loop de eventos de la app
app.exec()
```

BonusTrack: ¿Qué cosas copadas tiene PyQt?

Ventanas de diálogo para reutilizar

Existen algunos diálogos ya creados por el framework de Qt para que reutilicemos, como el **QFileDialog**, **QMessageBox**, **QColorDialog**, **QInputDialog**, **QProgressDialog**, entre otros... y su utilización es muy sencilla. Observar en la Fig. 26.
Se los puede utilizar configurandolos con el constructor, instanciándolos y luego llamando a alguno de sus métodos para hacerlos visibles y luego recuperar los datos que obtuvieron. Por otro lado, se pueden ejecutar desde alguno de sus métodos estáticos.

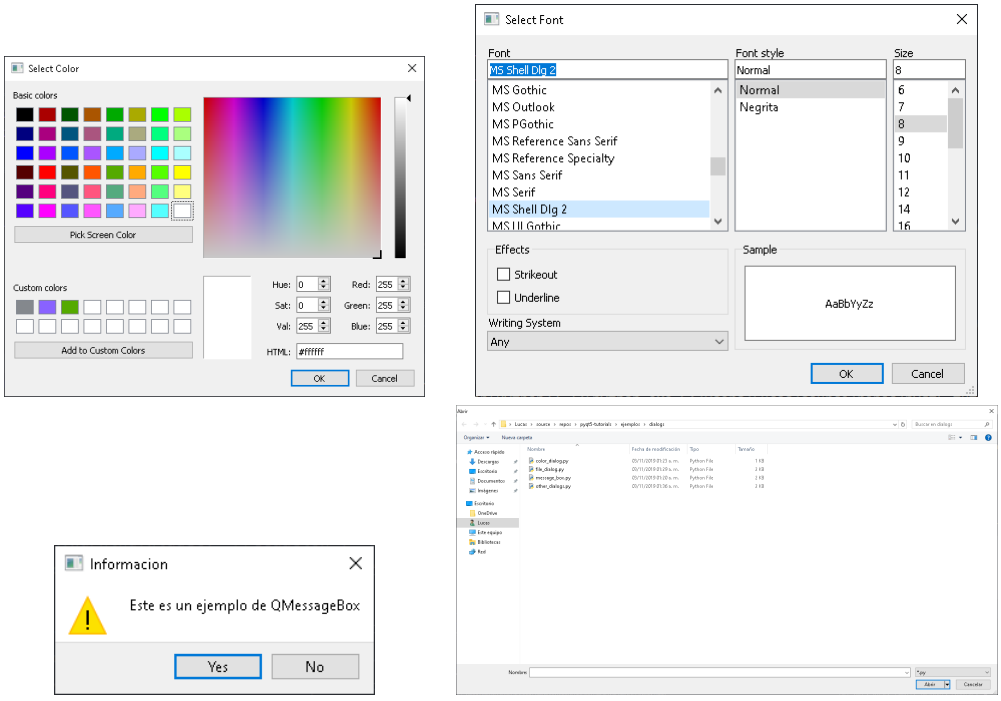


Figura 26: Ejemplos de QColorDialog, QFileDialog, QFontDialog y QMessageBox

PyQt

¿Cómo estructurar los proyectos?

Para estructurar los proyectos proponemos la siguiente estructura, obtenida inicialmente de lo que suele hacerse en PyQt5 y modificada con nuestra experiencia. Es una sugerencia.

/assets Archivos multimedia, u otro tipo de archivos, utilizados como recursos gráficos o como fuente de información estática.

/designer Archivos donde diseñamos el .ui en QtDesigner.

/resources Archivos de recursos creados en QtDesigner a partir de los assets.

/tests Si realizan pruebas, usando Unit Testing, o alguna otra tecnología, o crudo.

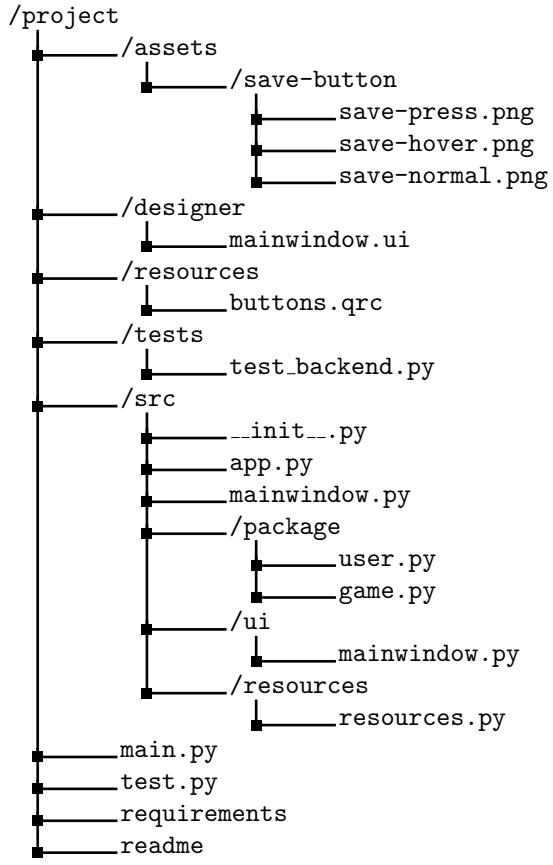
/src/package Código independiente de la interfaz gráfica.

/src/ui Archivos compilados a partir de los .ui del QtDesigner.

/src/resources Archivos compilados a partir de los .qrc del QtDesigner

/src/*.py Serán los archivos donde heredamos la interfaz compilada y conectamos la lógica.

/src/app.py Creamos una función de la aplicación donde se instancia el MainWindow y se ejecuta el loop de eventos, que será llamado por el main.py. Por lo general, adquiere relevancia cuando hay más conexiones a realizar, por ejemplo si el MainWindow necesitara conectarse a algún servicio que corre con otro módulo, o que se provee externamente cuando se la construye.



¿Qué son Signals, Slots?

Signals y Slots: Son un mecanismo de comunicación entre objetos del framework de Qt, de forma análoga a cómo sucede en el caso de eventos y callbacks, en cuyo caso las Signals son los eventos y los callbacks son los Slots registrados para tales eventos.
En primer lugar, podrían ya existir signals o slots dentro de una clase, o bien podríamos crearlos. Usualmente sucede que cualquier método o función dentro de Python puede ser tomado sin problema como un Slot en el proceso de conexión de dos objetos, pero no será identificado automáticamente como un Slot en otros contextos. Veamos a continuación, cómo podríamos crear en una clase un Signal y un Slot.

```
# PyQt5 import...
from PyQt5.QtCore import pyqtSignal
from PyQt5.QtCore import pyqtSlot
from PyQt5.QtCore import QObject

# Creamos nuestra clase cualquiera, es importante que sea QObject o cualquiera
# que haya heredado de QObject, para que pueda albergar Signals
class MyClass(QObject):

    # Declaro de forma estatica las se\~nales de la clase
    # y como argumento de pyqtSignal se pasa el tipo de dato que
    # se manda en la comunicacion del evento, podria ser una clase
    changed = pyqtSignal()
    progress = pyqtSignal(int)

    # Creando un metodo que queremos definir como Slot,
    # tenemos que usar el decorator para poder declararlo de tal forma
    @pyqtSlot(int)
    def set_value(self, value: int):
        # Hacer algo...
        pass
```

Asumamos que ahora tenemos un objeto con eventos, luego si queremos conectar nuestro callback existen dos mecanismos. El mecanismo largo implica que la conexión esté declarada en términos de **Objecto-Sender, Signal, Slot** y con esto se establece la conexión. La forma corta implica simplemente indicarle al evento que se le conecta un callback. A continuación un ejemplo de ambas sintáxis.

```
""" 1era Forma """
# Necesitamos haber realizado el import y luego la conexi\'on del
# sender y su signal "changed" con slot llamado on_changed
from PyQt5 import QtCore
QtCore.QObject.connect(sender, QtCore.SIGNAL("changed()"), on_changed)

""" 2do Forma """
# Le indicamos al objeto y a su signal, que conecte
# el slot para ejecutar cuando se emita el evento
sender.changed.connect(on_changed)
```

Finalmente, asumamos que nosotros somos los que queremos notificar del evento ocurrido, si tenemos un evento interno en nuestra clase para avisar que cambio el estado del objeto, entonces

```
# Dentro de mi clase, emito el evento "changed", que supongamos que declaramos
# con un mensaje para aclarar o comentar cosas...
self.changed.emit("Emitiendo este evento... llamara a todos los slots")
```

¿Qué son los Property?

Cuando tenemos una clase, las variables de la instancia las solemos llamar miembros o atributos, pero en la práctica se hace una diferencia respecto de las **Propiedades**, que son variables miembro o atributos de las instancias que son accesibles únicamente a través de métodos Getter y Setter. Es decir, una propiedad no necesariamente es una variable, simplemente engloba los método Getter y Setter que en su código pueden estar utilizando una variable miembro.

```
# Import PyQt5...
from PyQt5.QtCore import pyqtProperty
```

```
# Creamos los metodos para acceder internamente a las variables
# miembro, se podria ademas ejecutar otro codigo, o hacer validacion
# esa es la principal gracia de las Property!
def get_name(self):
    """ Name's getter """
    return self._name

def set_name(self, name: str):
    """ Name's setter """
    self._name = name

# Habiendo creado nuestra clase... luego de forma est\'atica se crea
# una propiedad para declararla, luego la maneja internamente para cada
# instancia PyQt
name = pyqtProperty(str, get_name, set_name)
```

Finalmente, trabajar externamente sobre la variable name, será de igual forma que como con una variable interna de la clase, pero durante el get y set de tal miembro, se ejecutarán los métodos asignados. Esto se suele usar para **Binding**, conexión entre el back-end y el front-end de una forma prolija simplemente haciendo una conexión entre una propiedad de una clase y su ViewController en la GUI.

¿Qué son los Stylesheets?

Los **Stylesheets** básicamente es una sintáxis que se emplea para describir el contenido de las propiedades de un Widget que definen el estilo del mismo, de forma tal que puede crearse un archivo con tal sintáxis o bien cargarlo directo desde un String, alterando las propiedades y el estilo del Widget.
En primer lugar, es importante tener en cuenta cómo funciona la sintáxis, además de un conocimiento de que la construcción de una GUI internamente se compone como un árbol de Widgets, según estas se vean autocontenidas en otras Widgets de más alto nivel. De esta forma, el Stylesheet puede configurar las propiedades del Widget a quien pertenece, o bien puede alterar las propiedades de los objetos que se contienen en esa Widget, según su tipo. Más sencillamente explicado, puedo cambiar el color de fondo de mi Widget, pero también puedo decirle que cambie el fondo de todos los Botones dentro de mi Widget.
¿Cómo modificar o configurar esto?, por un lado puede realizarse directamente desde QtDesigner si lo estamos haciendo en el momento para darle estilo a nuestra aplicación. Como se muestra en la Fig. 27

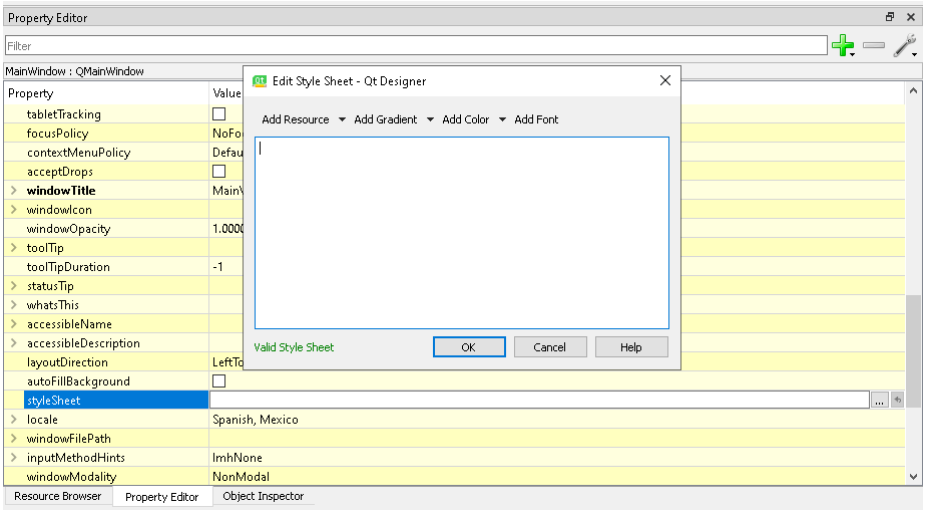


Figura 27: Configurar el estilo desde QtDesigner

Desde el código la primera forma de hacerlo sería.

```
my_widget.setStyleSheet("background-color: black;")
```

Otra forma más elegante y organizada ser'ia crear un archivo, supongamos **style.css** en donde albergamos todo el estilo que vamos a usar con la correcta sintáxis antedicha, entonces en el código...

```
    with open("styles.css") as f:
        my_widget.setStyleSheet(f.read())
```

Horizonte: ¿Que más puedo hacer?

Dado que no es posible llegar a cubrir todos los aspectos en la guía, por lo menos no en un tiempo limitado, todo aquello que sea considerado **de interés** sobre **de qué soy capaz** usando el framework Qt, irá enumerado a continuación motivando a que lo lean por su cuenta. Los links son para el framework original en C++, es necesario entender el concepto y luego el paralelismo es sencillo usando las referencias de PyQt.
Es importante aclarar, cualquier duda se pueden comunicar.

- Puedo crear mis propios widgets para QtDesigner
- Puedo hacer animaciones
- Puedo dibujar objetos y controlar interacciones o Ejemplos