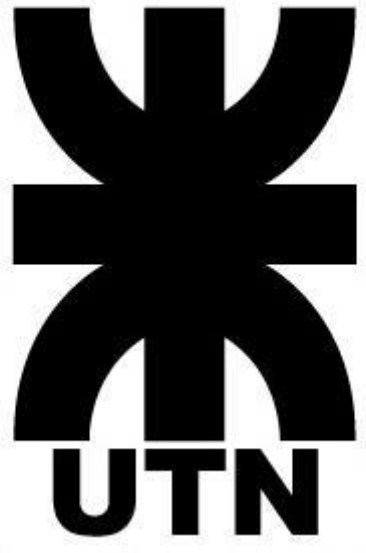
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL ROSARIO



Soporte a la Gestión de Bases de Datos con Programación Visual

Seminario: PyQt

Alumno	Legajo
Ambrosio, Facundo	49767
Coccoz, Manuel	49693
Condori Sosa, Juan	49499
Oldani, Marcos Alberto	50734

SEPTIEMBRE 2025

Índice

Introducción a Qt y PyQt

Qt es un framework de desarrollo de aplicaciones multiplataforma para desktop, embebido y móvil. Es utilizado para crear aplicaciones con interfaces modernas. Incluye una amplia variedad de componentes, tales como botones, menús, gráficos, formularios, tablas, entre muchos otros. Una de sus grandes ventajas es que permite escribir una interfaz de usuario que puede ejecutarse sin cambios en distintos sistemas operativos.

No es un lenguaje de programación por sí solo, sino que es un framework escrito en C++. Utiliza un preprocesador MOC (Meta-Object Compiler) para ampliar el lenguaje, agregando por ejemplo un mecanismo de signals y slots, que permite que los objetos del programa se comuniquen mediante eventos.

Qt es utilizado en aplicaciones populares tales como Adobe Photoshop Elements, Google Earth, TeamViewer, Telegram, y en empresas como la Agencia Espacial Europea, Dreamworks, LG, HP, Disney, y muchas más.

Además de todo lo anterior, Qt cuenta con bindings que permiten utilizar su framework en otros lenguajes, incluyendo Python mediante PySide y PyQt, que es el objeto de nuestro trabajo.

PyQt es un framework de Python que tiene como propósito el permitir que aplicaciones escritas en Python puedan aprovechar toda la funcionalidad de Qt haciendo uso de un SIP para el binding entre Python y C++. Tanto el SIP como PyQt son realizados por Riverbank Computing. PyQt al ser en esencia Qt tiene acceso a la mayoría de módulos de Qt como Qt Designer, QtSql, etc.

Los sistemas que soportan PyQt son Windows, macOS, Linux y algunas plataformas embebidas. PyQt aprovecha las características de hacer llamadas nativas al sistema anfitrión de Qt para asegurar que los widgets mantengan la apariencia y el comportamiento propios de cada plataforma.

En término de licencia PyQt tiene dos tipos dependiendo de su uso. Licencia GPL (GNU General Public License) que obliga la publicación del código fuente de la aplicación si se distribuye, y una licencia comercial que permite el desarrollo de software propietario cerrado, previo pago de licencia a Riverbank Computing.

Capacidades

- **Amplia colección de widgets**: Incluye widgets prediseñados para interfaces gráficas, como botones, etiquetas, campos de texto, menús desplegables,

pestañas, tablas, barras de progreso y mucho más. Estos son altamente personalizables y permiten construir interfaces profesionales sin necesidad de programar cada elemento desde cero.

- **Sistema de Signals y Slots**: usa el mecanismo de comunicación entre objetos propio de Qt. Un “signal” es un evento y un “slot” es la función que se ejecuta cuando esa “signal” ocurre. Esto simplifica la separación entre la lógica y la interfaz gráfica.
- **Compatibilidad multiplataforma**: las apps desarrolladas con PyQt pueden ejecutarse en Windows, Linux y macOS sin modificar el código. Qt se encarga de manejar las diferencias entre sistemas operativos, reduciendo tiempos de desarrollo y facilitando el deploy en distintos entornos.
- **Integración con Qt Designer**: es totalmente compatible con Qt Designer, una herramienta visual que permite crear interfaces gráficas mediante arrastrar y soltar componentes. Esto agiliza la construcción de interfaces y permite que diseñadores y desarrolladores trabajen en paralelo. Además de esta hay otras herramientas del ecosistema Qt compatibles con PyQt
- **Soporte para estilos y temas personalizados**: permite modificar la apariencia de los widgets usando Qt Style Sheets (QSS), una sintaxis similar a CSS que facilita la personalización de colores, fuentes, bordes y otros estilos visuales.
- **Módulos especializados**: incluye múltiples módulos que amplían sus capacidades más allá de la simple creación de ventanas, como QtMultimedia (reproducción de audio y video), QtWebEngine (integración de navegadores web), QtChart (creación de gráficos interactivos), QtSql (conexión y gestión de bases de datos SQL). Esto lo hace ideal para aplicaciones complejas y con múltiples funcionalidades.
- **Capacidades gráficas avanzadas**: ofrece herramientas para renderizar gráficos 2D y 3D, dibujar figuras personalizadas, manipular imágenes, aplicar transformaciones y trabajar con animaciones. Soporta integración con OpenGL, lo que permite desarrollar aplicaciones de visualización científica, modelado y simulación.
- **Internacionalización y localización**: Qt cuenta con herramientas para traducir aplicaciones a varios idiomas y adaptarlas a formatos de fecha, hora, moneda y escritura específicos de cada región. Esto permite que una aplicación desarrollada con PyQt sea utilizada por audiencias internacionales sin necesidad de rehacer la interfaz.

PyQt en profundidad

App básica

```
import sys

from PyQt6.QtCore import QSize, Qt
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

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

        self.setWindowTitle("App")
        self.setContentsMargins(10, 10, 10, 10)

        button = QPushButton("Esto es un botón")

        self.setFixedSize(QSize(400, 300))

        self.setCentralWidget(button)

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

Se crea una nueva clase que hereda de QMainWindow, el cual es un widget que provee la funcionalidad básica de una ventana principal. A esta le agregamos un título, un botón, ajustamos su tamaño, y fijamos al botón como widget central. A continuación se crea la aplicación, una instancia de la ventana, y se ejecuta.

Signals y Slots

Las señales son notificaciones emitidas por los widgets cuando ocurre algo. Ese “algo” puede ser muchas cosas: desde presionar un botón, hasta que cambie el texto en una caja de entrada, o que cambie el título de la ventana. Muchas señales son iniciadas por la acción del usuario, pero esto no es una regla. Además de notificar que algo ha ocurrido, las señales también pueden enviar datos para proporcionar contexto adicional sobre lo sucedido.

Los slots (ranuras) es el nombre que Qt utiliza para los receptores de las señales. Cualquier función (o método) puede usarse como un slot, conectando la señal a ella. Si la señal envía datos, la función receptora también recibe esos datos. Muchos widgets de Qt también tienen sus propios slots integrados, lo que significa que se pueden conectar widgets de Qt entre sí directamente.

```
import sys
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton

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

        self.setWindowTitle("App")
        self.setContentsMargins(10, 10, 10, 10)

        button = QPushButton("Esto es un botón")
        button.clicked.connect(self.button_clicked)

        self.setCentralWidget(button)

    def button_clicked(self):
        print("Botón presionado")

app = QApplication(sys.argv)

window = MainWindow()
window.show()

app.exec()
```

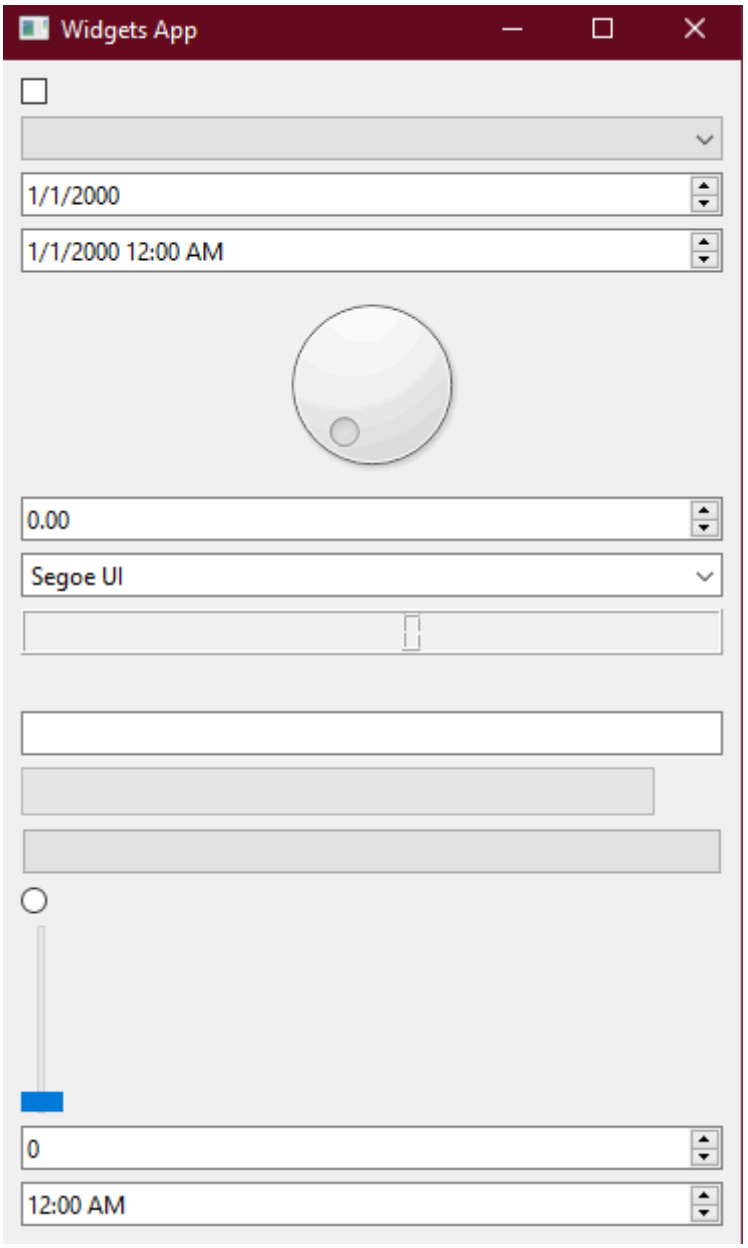
Se le agrega una señal al botón. Cada vez que se presiona, emite la señal `clicked`, conectada al método `button_clicked`.

Widgets y Layouts

Widget es el nombre que se le da a un componente de la interfaz con el que el usuario puede interactuar. Las interfaces de usuario están compuestas por múltiples widgets, organizados dentro de la ventana.

Qt incluye una amplia selección de widgets e incluso te permite crear tus propios widgets personalizados o personalizar los ya existentes.

```
import sys

from PyQt6.QtCore import Qt
from PyQt6.QtWidgets import QApplication, QCheckBox, QComboBox,
QDateEdit, QDateTimeEdit, QDial, QDoubleSpinBox, QFontComboBox, QLabel,
QLCDNumber, QLineEdit, QMainWindow, QProgressBar, QPushButton,
QRadioButton, QSlider, QSpinBox, QTimeEdit, QVBoxLayout, QWidget

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

        self.setWindowTitle("Widgets App")

        layout = QVBoxLayout()
        widgets = [QCheckBox, QComboBox, QDateEdit, QDateTimeEdit,
QDial, QDoubleSpinBox, QFontComboBox, QLCDNumber, QLabel, QLineEdit,
QProgressBar, QPushButton, QRadioButton, QSlider, QSpinBox, QTimeEdit]

        for w in widgets:
            layout.addWidget(w())

        widget = QWidget()
        widget.setLayout(layout)

        self.setCentralWidget(widget)

app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```



Para ordenar visualmente los widgets, se utilizan layouts. De los cuales hay cuatro:

- QHBoxLayout: Orden horizontal
- QVBoxLayout: Orden vertical
- QGridLayout: Forma de grilla con filas y columnas
- QStackedLayout: Permite ubicar widgets uno encima del otro

Threading

En Qt, la interfaz gráfica corre en un único hilo principal, que se encarga de dibujar la ventana y los widgets, responder a eventos de usuario, y actualizar la interfaz. El problema comienza al ejecutar una tarea pesada como la descarga de información o una consulta a base de datos, la interfaz se congela hasta que dicha tarea termine. La

solución es mover estas tareas a un hilo secundario, que envía señales una vez finaliza el trabajo.

```
import sys, time
from PyQt6.QtCore import QThread, pyqtSignal
from PyQt6.QtWidgets import QApplication, QMainWindow, QPushButton,
QLabel, QVBoxLayout, QWidget

# Worker
class Worker(QThread):
    finished = pyqtSignal(str)   # señal para avisar cuando termina

    def run(self):
        time.sleep(5)
        self.finished.emit("Tarea completada")

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Ejemplo con Threads")

        self.label = QLabel("Esperando...")
        self.button = QPushButton("Iniciar tarea")
        self.button.clicked.connect(self.start_task)

        layout = QVBoxLayout()
        layout.addWidget(self.label)
        layout.addWidget(self.button)

        container = QWidget()
        container.setLayout(layout)
        self.setCentralWidget(container)

    def start_task(self):
        self.label.setText("Trabajando...")
        self.button.setEnabled(False)

        # crear y arrancar el hilo
        self.worker = Worker()
        self.worker.finished.connect(self.task_done)
```

```
        self.worker.start()

    def task_done(self, message):
        self.label.setText(message)
        self.button.setEnabled(True)

app = QApplication(sys.argv)
window = MainWindow()
window.show()
app.exec()
```

Al presionar el botón, se inicia `Worker` en un hilo aparte, al terminar emite la señal `finished`, y se actualiza el `QLabel`.

Arquitectura Model/View

Qt utiliza una arquitectura de Model/View para gestionar y mostrar datos. El modelo contiene los datos y los organiza, mientras que la vista es esencialmente el widget que muestra los datos en lista, tabla o árbol. También existe un Delegate (delegado), que se encarga de cómo se renderizan los ítems y como se editan. El flujo normal es el siguiente:

- La vista solicita al modelo los datos para mostrar
- El delegado decide cómo se renderizan los datos
- Cuando el usuario edita algo, la vista notifica al modelo para actualizar los datos

La vista hace el llamado a métodos específicos como rowCount(), columnCount() y data().

Diferencia con el patrón MVC: Este tiene tres componentes: El modelo que maneja los datos, la vista que los muestra, y el controlador que gestiona las acciones del usuario. En la arquitectura Model/View, no existe un controlador, la responsabilidad de éste se reparte entre la vista y el delegado. La vista maneja interacciones de usuario, el delegado maneja el renderizado y la edición.

```
from PyQt6.QtCore import Qt, QAbstractListModel
from PyQt6.QtWidgets import QApplication, QListView
import sys

class NameModel(QAbstractListModel):
    def __init__(self, names=None):
        super().__init__()
        self.names = names or []
```

```
    def data(self, index, role):
        if role == Qt.ItemDataRole.DisplayRole:
            return self.names[index.row()]

    def rowCount(self, index):
        return len(self.names)

app = QApplication(sys.argv)

model = NameModel(["Nombre 1", "Nombre 2", "Nombre 3"])
view = QListView()
view.setModel(model)

view.show()
app.exec()
```

En este ejemplo, el modelo `NameModel` guarda los nombres, y la vista `QListView` los muestra.

Qt Designer y QSS

Qt Designer es una herramienta para diseñar interfaces de usuario de manera visual, ahorrando la necesidad de escribir código a mano. Permite arrastrar y soltar widgets, configurar propiedades tales como texto, tamaños, iconos, alineación, etc, definir layouts, y exportar la interfaz en formato XML. Estos archivos luego pueden cargarse a Python o bien convertirse en código Python. La ventaja obvia es que separa el diseño de la lógica. Para gestionar la apariencia visual, Qt utiliza QSS (Qt Style Sheets), que es muy similar a CSS.

```
QPushButton {
    background-color: gray;
    color: white;
    font-size: 16px;
    border-radius: 8px;
    padding: 6px;
}
QPushButton:hover {
    background-color: green;
}
```

Comparación con tecnologías similares

Aspecto	PyQt	PySide	Kivy	Eel
Descripción	Biblioteca basada en Qt para interfaces gráficas. Robusta y ampliamente utilizada.	Alternativa a PyQt, también basada en Qt, mantenida por The Qt Company.	Framework multiplataforma para aplicaciones gráficas con interfaz moderna.	Biblioteca ligera para crear aplicaciones de escritorio con HTML/CSS/JS.
Licencia	GPL o comercial (pago para uso comercial).	LGPL (más flexible para uso comercial).	MIT (libre y de código abierto).	MIT (libre y de código abierto).
Facilidad de Uso	Moderada. Curva de aprendizaje alta por su complejidad y extensa API.	Similar a PyQt, pero con documentación poco clara en algunos casos.	Moderada. Sintaxis propia (lenguaje KV) que puede requerir aprendizaje.	Muy fácil. Ideal para desarrolladores web que conocen HTML/CSS/JS.
Rendimiento	Excelente para aplicaciones de escritorio complejas.	Rendimiento comparable a PyQt, optimizado para aplicaciones de escritorio.	Bueno para aplicaciones gráficas y móviles, pero puede ser más lento en escritorio.	Bueno para **aplicaciones ligeras**, pero limitado por el motor del navegador.
Compatibilidad	Windows, macOS, Linux. Soporta aplicaciones de escritorio robustas.	Windows, macOS, Linux. Similar a PyQt, con mejor integración en algunos casos.	Windows, macOS, Linux, Android, iOS. Ideal para multiplataforma.	Windows, macOS, Linux. Depende de un navegador Chromium.
Soporte Móvil	Limitado, requiere Qt for Mobile (complejo).	Similar a PyQt, con soporte limitado para móvil.	Excelente soporte para Android e iOS.	No diseñado para móviles, enfocado en escritorio.

Aspecto	PyQt	PySide	Kivy	Eel
Casos de Uso	Aplicaciones de escritorio complejas (ej. IDEs, editores gráficos).	Similar a PyQt, pero preferido en proyectos comerciales por su licencia.	Aplicaciones móviles, juegos, interfaces táctiles.	Aplicaciones ligeras de escritorio con interfaz web.
Comunidad y Soporte	Gran comunidad, pero la documentación es muy técnica.	Comunidad activa, soporte oficial de Qt.	Comunidad activa, buena documentación para principiantes.	Comunidad pequeña. **Actualmente abandonado.**
Integración con Web	Limitada, requiere módulos adicionales como QtWebEngine.	Similar a PyQt, con soporte para QtWebEngine.	No está diseñada para integración web directa.	Excelente, usa HTML/CSS/JS como frontend y Python como backend.
Tamaño del Ejecutable	Grande, debido a las dependencias de Qt.	Similar a PyQt, ejecutables grandes.	Moderado, pero puede aumentar con dependencias para móviles.	Ligero, aunque depende de incluir un navegador Chromium.
Personalización UI	Alta, con Qt Designer para diseño visual de interfaces.	Similar a PyQt, compatible con Qt Designer.	Alta, con lenguaje KV y widgets personalizables.	Muy alta, usa tecnologías web (CSS, frameworks como Tailwind).

En resumen:

- **PyQt**: Ideal para aplicaciones de escritorio robustas y complejas, pero con restricciones de licencia para uso comercial.
- **PySide**: Similar a PyQt, pero con una licencia más flexible (LGPL), lo que lo hace preferible para proyectos comerciales.
- **Kivy**: Perfecto para aplicaciones multiplataforma, especialmente móviles, con interfaces modernas y táctiles.
- **Eel**: Excelente para desarrolladores web que desean crear aplicaciones de escritorio ligeras con tecnologías web.

Una comparación de descargas puede encontrarse en:
https://www.piptrends.com/compare/pyqt6-vs-pyside6-vs-kivy

Utilización

PyQt, al ser una herramienta para desarrollar interfaces gráficas, presenta un uso muy variado. Podemos citar algunos casos puntuales:

- **Dropbox**: Plataforma de almacenamiento en la nube, permite a los usuarios almacenar y sincronizar archivos en línea y entre ordenadores y compartir archivos y carpetas con otros usuarios. La aplicación de escritorio utilizó PyQt en sus primeras versiones, lo que permitió un desarrollo rápido y fluido en esas primeras etapas. Con el tiempo, migró hacia otras tecnologías.
- **Spyder**: Es un entorno de desarrollo open source pensado para ser utilizado por científicos, ingenieros, y analistas de datos. Posee funciones avanzadas de editado, análisis, debugging y perfilado de código.
- **Veusz**: Una herramienta para crear gráficos científicos listos para publicar, en dos o tres dimensiones. Busca lograr que la interfaz sea lo más simple y consistente posible.
- **Eric IDE**: Otro entorno de desarrollo para Python, escrito totalmente en Python, multiplataforma. Ofrece resaltado de sintaxis, autocompletado, debugging, resaltado de errores, funciones de búsqueda, control de versiones, soporte para tests unitarios, entornos virtuales, traducciones en varios idiomas, plugins, y más.

Demanda

La demanda laboral de PyQt existe, pero es muy acotada en comparación con otras tecnologías mas generalistas. En general, hay más empresas buscando conocimiento de Qt que PyQt o PySide, quizás por el hecho de que C++ es un lenguaje con mejor rendimiento. En las vacantes que piden conocimiento en Python, PyQt no se pide como un requerimiento excluyente, más bien como un complemento. En el caso de Argentina no se pudo encontrar ningún puesto que pida conocimiento en PyQt.

Un factor que consideramos importante para explicar (en parte) lo anterior es la licencia. Muchas empresas optan por elegir PySide, ya que no es necesario pagar por una licencia comercial, que tiene un costo de **U$D 670** por cada desarrollador. Esto ha hecho que PyQt se vea más frecuentemente en proyectos open source, en aplicaciones académicas, científicas o educativas, y en herramientas desarrolladas por la comunidad.

Código de ejemplo

Para demostrar la funcionalidad básica de PyQt se ha construido una aplicación simple, que consiste de una página principal, una calculadora, un bloc de notas, y un visor de

imágenes, todo accesible mediante una barra de navegación, estilizado con un tema personalizado.

El repositorio se puede encontrar en:
https://github.com/oldaniMarcos/Soporte-Seminario-PyQt

Bibliografía

- https://riverbankcomputing.com/software/pyqt/intro
- https://www.qt.io/
- https://pypi.org/project/PyQt6/
- https://wiki.qt.io/About_Qt/es
- https://github.com/python-eel/Eel
- https://wiki.qt.io/Qt_for_Python
- https://kivy.org/
- https://news.ycombinator.com/item?id=32245091
- https://en.wikipedia.org/wiki/Dropbox
- https://www.spyder-ide.org/
- https://veusz.github.io/
- https://eric-ide.python-projects.org/