Using Qt with pyside6:

At minimum:

Core run time:
- Qapplication
- Qt event loop

Windowing:
- Qwidget
- Qlayout (QVBoxLayout, QHBoxLayout)

Widgets:
- QPushButton
- Qcombobox
- QTextEdit
- Qlabel
- Qmessage box

Events:
- Signals & slots

Threads:
- Threads.

Loop system:

This is python and not C so the main loop isn't declared explicitly, but written in the code as sys.exit(app.exec())
Internally Qt makes it run:

```
while (application running) {
   wait_for_event();
   dispatch_event();
}
```

This loop: (If blocked OS freezes)

- Waits for mouse clicks
- waits for Key presses
- Waits for timers
- Waits for OS events

Creating a GUI step by step with inputs and outputs for embedded applications

This is a guide in python 8to easily create a interface which can operate in a Linux OS given the open source drivers are installed. This is a powerful set of tools to use since it allows to make a interface to robotics or instrumentation interface.

GUI_BUTTONS.py – L.S.D

```
import sys
from PySide6.QtWidgets import QApplication, QWidget, QPushButton
class MyWindow(QWidget):
    def __init__(self):

        super().__init__()
        self.setWindowTitle("Qt Geometry Learning")
        self.setFixedSize(600, 600)

        self.btn1 = QPushButton("Button 1\npos=(20,20)\nsize=(120x50)", self)
        self.btn1.move(20, 20)        # x, y
        self.btn1.resize(120, 50)     # width, height
        self.btn1.clicked.connect(self.on_btn1)

    def on_btn1(self):
        print("Button 1 pressed")

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

```
# import Qt

# define class
# initiate self -> class Mywindow

#super() init
# self.WindowTitle(“text”)
# self.setFixedSize(600,600)

# self.btn1 = QpushButton(“text”, self)

# self.bnt1.move(x, y)
# self.btn1.resize(w, h)
# self.btn1.clicked.connect(FUNC)


If__name__= “__main__”
(makes sure the code runs only when the file is executed directly not when imported).

App = Qapplication(sys.argv)
Window = MyWindow(()
Window.show
Sys.exit(app.exec())
```

Using the code descripted above a simple GUI can be made which is rather limited, this GUI has button inputs which link them to a print() statement, this click can be linked to any operation such as a subprocess output which executes a bash file or a compiled C script as can commonly be seen using higher level languages to have access to lower level compiled machine code executions.

Key:

- super().__init__() → **“initialize the QWidget part of my object”**
- __name__ == "__main__" → **“only run this when I launch this file”**

GUI_BUTTONS_IN_OUT.py - L.S.D

```
import sys
from PySide6.QtWidgets import QApplication, QWidget, QPushButton, QLineEdit, QLabel
class MyWindow(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Qt Input → Output Example")
        self.setFixedSize(600, 600)

        self.input_box = QLineEdit(self)
        self.input_box.setPlaceholderText("Enter a number")
        self.input_box.move(20, 20)
        self.input_box.resize(200, 20)

        self.btn1 = QPushButton("Submit", self)
        self.btn1.move(240, 20)
        self.btn1.resize(120, 40)
        self.btn1.clicked.connect(self.on_btn1)

        self.output_label = QLabel("Output will appear here", self)
        self.output_label.move(20, 80)
        self.output_label.resize(340, 40)

    def on_btn1(self):
        text = self.input_box.text()     # read input
        self.output_label.setText(text)   # display output
        print(f"Input received: {text}")

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

```
# import Qt

# define class
# initiate self -> class Mywindow

#super() init
# self.WindowTitle(“text”)
# self.setFixedSize(600,600)


# self.input_box = QlineEdit(self)
# self.input_box.setPlaceholderText("Enter a number")
# self.btn1 = QpushButton(“text”, self)


If__name__= “__main__”
(makes sure the code runs only when the file is executed directly not when imported).

// so in a more practical sense for single action user inputs it can be defined only using these and can be expanded to multiple inputs with no interrupt handling:

Key points:

- Define input & format  QLineEdit
- Link action to function connect(self.on_btn1)
- Link function to output QLabel

So the button is defined and input box is defined and both are visible in the UI, the function on_btn1 to link the value n the input to a output executed by the btn1.
```

So as can be seen this code is similar to the previous only with extra buttons, this can be used as a guideline to setting the buttons for the inputs. Its recommended to try moving around buttons and changing the widget size to get the hang of it. Its rather simple doesn’t require much understanding to get working reliably.

PlotWidget.py L.S.D

```
# PlotWidget.py BY Lorenzo Daidone - MIT

import sys
from PySide6.QtWidgets import (
    QApplication, QWidget, QPushButton,
    QLineEdit, QLabel
)
from PySide6.QtCharts import QChart, QChartView, QScatterSeries
from PySide6.QtCore import Qt

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()

        # ---- Window setup ----
        self.setWindowTitle("XY Plot Example")
        self.setFixedSize(600, 600)

        # ---- X input ----
        self.input_x = QLineEdit(self)
        self.input_x.setPlaceholderText("X value")
        self.input_x.move(20, 20)
        self.input_x.resize(100, 30)

        # ---- Y input ----
        self.input_y = QLineEdit(self)
        self.input_y.setPlaceholderText("Y value")
        self.input_y.move(140, 20)
        self.input_y.resize(100, 30)

        # ---- Button ----
        self.btn1 = QPushButton("Plot", self)
        self.btn1.move(260, 20)
        self.btn1.resize(100, 30)
        self.btn1.clicked.connect(self.on_btn1)

        # ---- Status label ----
        self.output_label = QLabel("Enter X and Y, then press Plot", self)
        self.output_label.move(20, 60)
        self.output_label.resize(340, 30)

        # ---- Plot setup ----
        self.series = QScatterSeries()
        self.series.setMarkerSize(10.0)

        self.chart = QChart()
        self.chart.addSeries(self.series)
        self.chart.createDefaultAxes()
        self.chart.setTitle("XY Plot")

        self.chart_view = QChartView(self.chart, self)
        self.chart_view.setRenderHint(self.chart_view.renderHints())
        self.chart_view.move(20, 100)
        self.chart_view.resize(560, 480)

    # ---- Button callback ----
    def on_btn1(self):
        try:
            x = float(self.input_x.text())
            y = float(self.input_y.text())

            self.series.append(x, y)
            self.output_label.setText(f"Plotted point: ({x}, {y})")
            print(f"Plotted: x={x}, y={y}")

        except ValueError:
            self.output_label.setText("Invalid input (enter numbers)")

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

For a more useful system in this code the values get plotted, for a simpler work flow and less variables there will be a 2 inputs one defined as Y and one defined as X when the button is pressed the values are passed to a function which takes those values as a input for Qt widget display function. In this case its Qcharts functions.

#defenitions (as before + Qtcharts)

define class
initiate self -> class Mywindow

#super() init
self.**WindowTitle**(“text”)
self.**setFixedSize**(600,600)

X input only definition in code changes for maintainability

Y input same as X but location is different

status label

Plot setup
self is the object used to define the scatter dots QscatterSeries
here we set a Qt function to the object for the scatter

chart object from class self, Qchart Qt knows now it’s a chart.
self.chart.**addSeries**(self.series) <- adds the values in the scatter to the chart

#self.chart.setTitle() here the title is set.

QchartView() <- here is how its viewed

use .move() move the chart
resize to set the sizing

This code is a good example for using inputs & outputs in the GUi, the main issue is everything is within the Qt thread meaning functions like input() which take a terminal input cannot be used in the QT thread like used in this code input since this code inputs are within the thread of the QT widget.

GUI_TERMINAL_INPUT.py – L.S.D

```
# GUI_TERMINAL_INPUT.py
import sys
from PySide6.QtWidgets import QApplication, QWidget, QPushButton, QLabel
from PySide6.QtCore import QThread, Signal

# ---- Thread that waits for terminal input ----
class TerminalInputThread(QThread):
    text_received = Signal(str)

    def run(self):
        text = input("Enter something in terminal: ")
        self.text_received.emit(text)

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()

        # ---- Window setup ----
        self.setWindowTitle("Terminal Input (Threaded)")
        self.setFixedSize(400, 200)

        # ---- Button ----
        self.btn = QPushButton("Get input from terminal", self)
        self.btn.move(20, 20)
        self.btn.resize(200, 40)
        self.btn.clicked.connect(self.on_button_pressed)

        # ---- Output label ----
        self.label = QLabel("Waiting for input...", self)
        self.label.move(20, 80)
        self.label.resize(360, 40)

        self.input_thread = None

    def on_button_pressed(self):
        print("Button pressed → waiting for terminal input")
        self.label.setText("Check terminal and type input...")

        self.input_thread = TerminalInputThread()
        self.input_thread.text_received.connect(self.update_label)
        self.input_thread.start()

    def update_label(self, text):
        self.label.setText(f"Received: {text}")
        print(f"Received from terminal: {text}")

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

import QThread and previous modules

New thread define as class Terminal Input
Text received = signal from terminal

define the functions linked to this thread

set up visuals as before

function once button pressed

get value

MAIN POINT:

GUI thread → buttons, display
Worker thread → blocking I/O
Signal → safe communication

- continuous terminal input
- stopping threads cleanly
- replacing input() with serial/socket
- adding a queue instead of single values

Threads_continous.py

```
import sys
import time
from PySide6.QtWidgets import QApplication, QWidget, QPushButton, QLabel
from PySide6.QtCore import QThread, Signal

class Worker(QThread):
    value = Signal(int)

    def run(self):
        i = 0
        while True:
            time.sleep(0.5)     # blocking here is OK
            self.value.emit(i)   # send data to GUI
            i += 1

class MyWindow(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Continuous Data Demo")
        self.setFixedSize(300, 150)

        self.label = QLabel("Value: 0", self)
        self.label.move(20, 20)
        self.label.resize(200, 30)

        self.button = QPushButton("Click me", self)
        self.button.move(20, 60)
        self.button.resize(120, 30)
        self.button.clicked.connect(self.on_click)

        self.worker = Worker()
        self.worker.value.connect(self.update_label)
        self.worker.start()

    def update_label(self, v):
        self.label.setText(f"Value: {v}")

    def on_click(self):
        print("Button still works!")

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

```
# import as before

# define a new Qthread as work class, this will the continuous loop
# define the object (value)

# here the function is made
# continuous adder

# a sleep in this thread only affects this scope
# here the value updates the interface

# defined class as window

# link value to object of class worker to my window(self)
```

This code demonstrates how to keep a constant thread running while having the GUI functional, this can be used in many cases such as streaming in data from a microcontroller or running other scripts which link to this one via a terminal input.

Main points:

- define a Qthread as a class
- link object through emit
- define class in window class
- update via function in window class

MCU serial communication: (Serial Module)

SERIAL_SIMPLEST.py

```
import serial                                                    # import serial <- used to get serial data (USB)

ser = serial.Serial(                                             #define the port baud.. etc
    port="/dev/cu.usbmodem111401",
    baudrate=115200,
    timeout=1
)
print("Connected")

while True:                                                      While(1) line = data on serial line (UTF-8)
    line = ser.readline().decode(errors="ignore").strip()        # 3 operations chained together to read bytes from serial port <- IMPORTANT
    if line:
        print(line)
```

- **Ser.readline()** # reads from port stops for new lines (MCU -> \n)
- **.decode(errors = "ignore")** # turns UTF-8 into python Unicode strings, ignore in case of corrupted data
- **.strip**() # removes trailing spaces for cleaner prints

SERIAL_SELECT_IN_SIMPLE

```
import serial                                                    # Serial + serial tools import
import serial.tools.list_ports


# ---- List available ports ----
ports = list(serial.tools.list_ports.comports())                 # define the list not the port

if not ports:
    print("No serial ports found")                               # debug print statement
    exit(1)

print("Available serial ports:")                                 # print avail ports statement
for i, p in enumerate(ports):
    print(f"{i}: {p.device}")                                    # actual ports

# ---- Select port ----
idx = int(input("Select port number: "))                         # input() have a input port
port = ports[idx].device                                         # ports[idx].device <- here it connects

# ---- Open serial ----
ser = serial.Serial(port, 115200, timeout=1)                     # setup dig parameters
print(f"Connected to {port}")

# ---- Read loop ----
while True:                                                      #Read loop
    line = ser.readline().decode(errors="ignore").strip()
    if line:
        print(line)
```

This is rather important since it allows to use a usb as a input to be used in any python program, this!!! Is very useful and I will have a specified file to also describe the drivers used.

C

Connecting Pyside6 with serial:

GUI_MCU_PRINT.py pt1

```
import sys
import serial
import serial.tools.list_ports
from PySide6.QtWidgets import (                                  # import QT and Serial
    QApplication, QWidget, QVBoxLayout, QHBoxLayout,
    QPushButton, QTextEdit, QComboBox, QLabel, QMessageBox
)
from PySide6.QtCore import QThread, Signal
# ---------- Serial Thread ----------
class SerialReader(QThread):                                     # create the signal thread
    data_received = Signal(str)
    error = Signal(str)

    def __init__(self, port, baudrate=115200):                   # define port baud
        super().__init__()                                       # sine the port changes
        self.port = port
        self.baudrate = baudrate
        self.running = True
        self.ser = None

    def run(self):
        try:
            self.ser = serial.Serial(self.port, self.baudrate, timeout=1)
            while self.running:
                line = self.ser.readline().decode(errors="ignore").strip()
                if line:
                    self.data_received.emit(line)
        except Exception as e:
            self.error.emit(str(e))
        finally:
            if self.ser and self.ser.is_open:
                self.ser.close()

    def stop(self):
        self.running = False
        self.wait()
```

GUI_MCU_PRINT.py pt1

```
# ---------- GUI ----------
class SerialMonitor(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Simple MCU Serial Monitor")
        self.setFixedSize(600, 400)
        self.reader = None
        self.init_ui()
        self.refresh_ports()

    def init_ui(self):
        layout = QVBoxLayout(self)

        # ---- Port row ----
        port_row = QHBoxLayout()
        port_row.addWidget(QLabel("Serial Port:"))
        self.port_combo = QComboBox()
        port_row.addWidget(self.port_combo)

        self.refresh_btn = QPushButton("Refresh")
        self.refresh_btn.clicked.connect(self.refresh_ports)
        port_row.addWidget(self.refresh_btn)
        layout.addLayout(port_row)

        # ---- Connect buttons ----
        btn_row = QHBoxLayout()
        self.connect_btn = QPushButton("Connect")
        self.connect_btn.clicked.connect(self.connect_serial)
        btn_row.addWidget(self.connect_btn)
        self.disconnect_btn = QPushButton("Disconnect")
        self.disconnect_btn.clicked.connect(self.disconnect_serial)
        self.disconnect_btn.setEnabled(False)
        btn_row.addWidget(self.disconnect_btn)
        layout.addLayout(btn_row)

        # ---- Output ----
        self.output = QTextEdit()
        self.output.setReadOnly(True)
        layout.addWidget(self.output)
```

GUI_MCU_PRINT.py pt3

```
    # -------- Logic --------
    def refresh_ports(self):
        self.port_combo.clear()
        for p in serial.tools.list_ports.comports():
            self.port_combo.addItem(p.device)

    def connect_serial(self):
        if self.reader:
            return

        port = self.port_combo.currentText()
        if not port:
            QMessageBox.warning(self, "No port", "Select a serial port")
            return

        self.reader = SerialReader(port)
        self.reader.data_received.connect(self.output.append)
        self.reader.error.connect(self.on_error)
        self.reader.start()

        self.output.append(f"Connected to {port}")
        self.connect_btn.setEnabled(False)
        self.disconnect_btn.setEnabled(True)

    def disconnect_serial(self):
        if self.reader:
            self.reader.stop()
            self.reader = None
            self.output.append("Disconnected")

        self.connect_btn.setEnabled(True)
        self.disconnect_btn.setEnabled(False)

    def on_error(self, msg):
        self.output.append(f"[ERROR] {msg}")
        self.disconnect_serial()

    def closeEvent(self, event):
        self.disconnect_serial()
        event.accept()

# ---------- Main ----------
if __name__ == "__main__":
    app = QApplication(sys.argv)
    win = SerialMonitor()
    win.show()
    sys.exit(app.exec())
```