Plotting with PyQtGraph

Create custom plots in PyQt with PyQtGraph

by John Lim Read time 14:16 Graphics and Plotting

Graphics and Plotting course

- Plotting with PyQtGraph
- Plotting with Matplotlib

One of the major strengths of Python is in exploratory data science and visualization, using tools such as Pandas, numpy, sklearn for data analysis and matplotlib plotting. Buiding GUI applications with PyQt gives you access to all these Python tools directly from within your app, allowing you to build complex data-driven apps and interactive dashboards.

While it is possible to embed `matplotlib` plots in PyQt the experience does not feel entirely *native*. For simple and highly interactive plots you may want to consider using PyQtGraph instead. PyQtGraph is built on top of PyQ5 native `QGraphicsScene` giving better drawing performance, particularly for live data, as well as providing interactivity and the ability to easily customize plots with Qt graphics widgets.

In this tutorial we'll walk through the first steps of creating a plot widget with PyQtGraph and then demonstrate plot customization using line colours, line type, axis labels, background colour and plotting multiple lines.

Getting started

To be able to use PyQtGraph with PyQt you first need to install the package to your Python environment. You can do this as normal using `pip`.

BASH

```
pip install pyqtgraph
```

Once the installation is complete you should be able to import the module as normal.

Creating a PyQtGraph widget

In PyQtGraph all plots are created using the `PlotWidget` widget. This widget provides a contained *canvas* on which plots of any type can be added and configured. Under the hood, this plot widget uses Qt native `QGraphicsScene` meaning it fast and efficient yet simple to integrate with the rest of your app. You can create a `PlotWidget` as for any other widget.

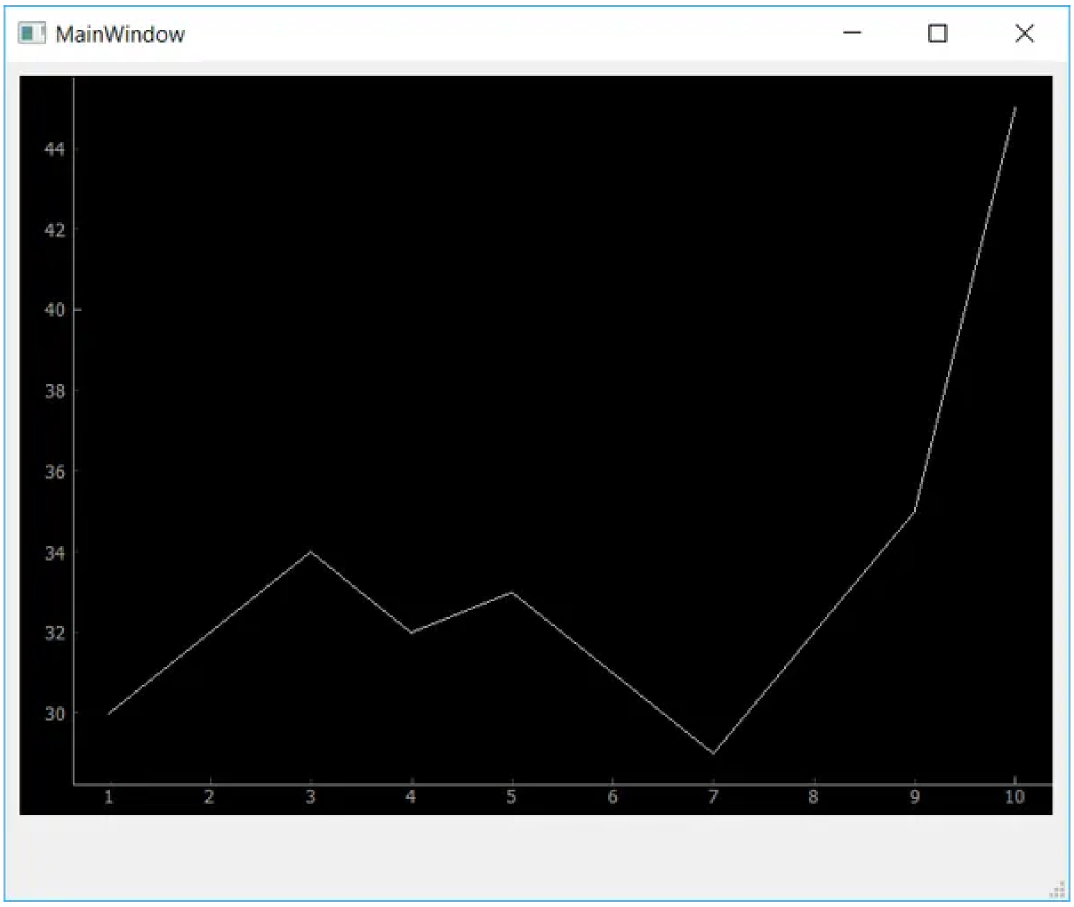
The basic template app, with a single `PlotWidget` in a `QMainWindow` is shown below.

In the following examples we'll create the PyQtGraph widget in code. Want to know how to embed PyQtGraph when using Qt Designer? See Embedding custom widgets from Qt Designer

PYTHON

```python
from PyQt5 import QtWidgets
from pyqtgraph import PlotWidget, plot
import pyqtgraph as pg
import sys  # We need sys so that we can pass argv to QApplication
import os

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1,2,3,4,5,6,7,8,9,10]
        temperature = [30,32,34,32,33,31,29,32,35,45]

        # plot data: x, y values
        self.graphWidget.plot(hour, temperature)


def main():
    app = QtWidgets.QApplication(sys.argv)
    main = MainWindow()
    main.show()
    sys.exit(app.exec_())


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

In all our examples below we import PyQtGraph using `import pyqtgraph as pg`. This is a common convention in PyQtGraph examples to keep things tidy & reduce typing. You an import and use it as `import pyqtgraph` if you prefer.



The custom PyQtGraph widget showing dummy data.

The default plot style of PyQtGraph is quite bare — a black background with a thin (barely visible) white line. In the next section we'll look at what options we have available to us in PyQtGraph to improve the appearance and usability of our plots.

Styling plots

PyQtGraph uses Qt's `QGraphicsScene` to render the graphs. This gives us access to all the standard Qt line and shape styling options for use in plots. However, PyQtGraph provides an API for using these to draw plots and manage the plot canvas.

Below we'll go through the most common styling features you'll need to create and customize your own plots.

Background Colour

Beginning with the app skeleton above, we can change the background colour by calling `.setBackground` on our `PlotWidget` instance (in `self.graphWidget`). The code below will set the background to white, by passing in the string 'w'.

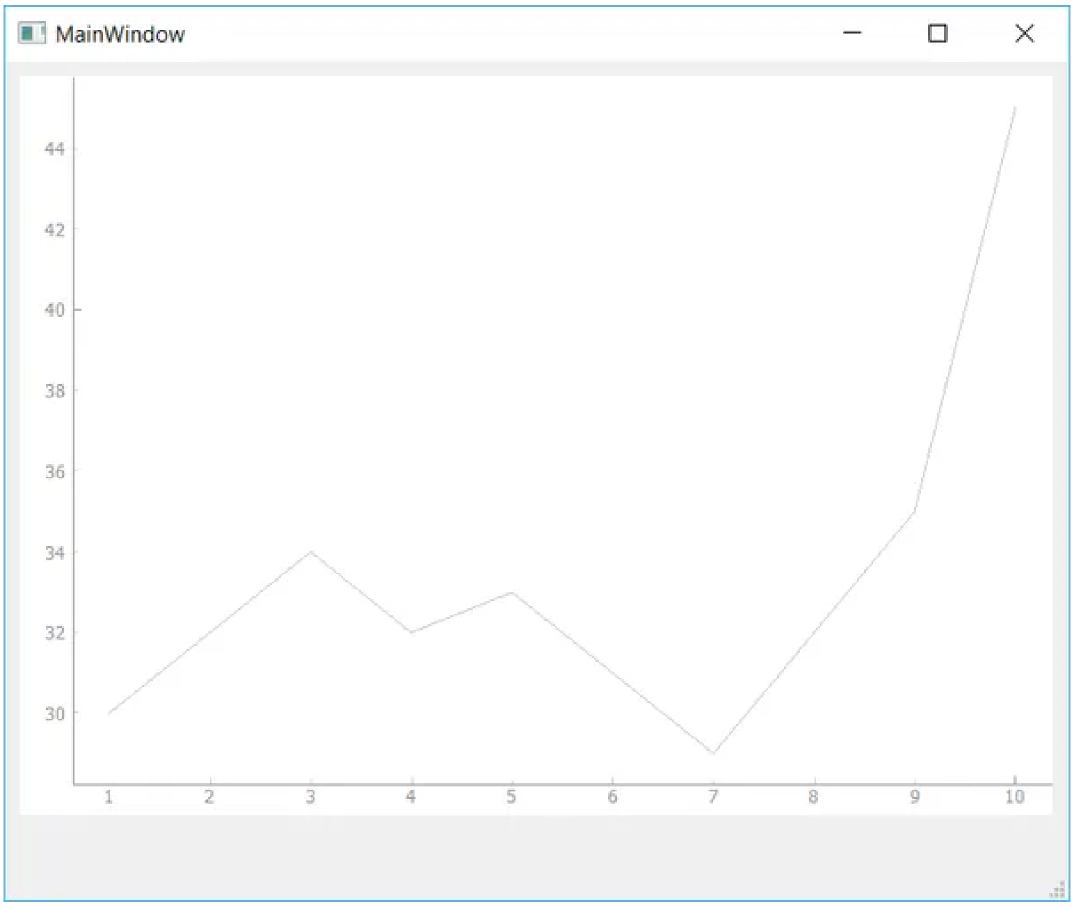
PYTHON

```python
self.graphWidget.setBackground('w')
```

You can set (and update) the background colour of the plot at any time.

PYTHON

```
from PyQt5 import QtWidgets
from pyqtgraph import PlotWidget, plot
import pyqtgraph as pg
import sys  # We need sys so that we can pass argv to QApplication
import os

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1,2,3,4,5,6,7,8,9,10]
        temperature = [30,32,34,32,33,31,29,32,35,45]

        self.graphWidget.setBackground('w')
        self.graphWidget.plot(hour, temperature)


def main():
    app = QtWidgets.QApplication(sys.argv)
    main = MainWindow()
    main.show()
    sys.exit(app.exec_())


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



Change PyQtGraph Plot Background to White

There are a number of simple colours available using single letters, based on the standard colours used in `matplotlib`. They're pretty unsurprising, except that 'k' is used for black.

In addition to these single letter codes, you can also set more complex colours using hex notation eg. #672922 as a string.

PYTHON

```python
self.graphWidget.setBackground('#bbccaa')         # hex
```

RGB and RGBA values can be passed in as a 3-tuple or 4-tuple respectively, using values 0-255.

PYTHON

```python
self.graphWidget.setBackground((100,50,255))      # RGB each 0-255
self.graphWidget.setBackground((100,50,255,25))   # RGBA (A = alpha opacity)
```

Lastly, you can also specify colours using Qt's `QColor` type directly.

PYTHON

```python
from PyQt5 import QtGui  # Place this at the top of your file.
self.graphWidget.setBackground(QtGui.QColor(100,50,254,25))
```

This can be useful if you're using specific `QColor` objects elsewhere in your application, or to set your plot background to the default GUI background colour.

PYTHON

```python
color = self.palette().color(QtGui.QPalette.Window)  # Get the default window background,
self.graphWidget.setBackground(color)
```

Line Colour, Width & Style

Lines in PyQtGraph are drawn using standard Qt `QPen` types. This gives you the same full control over line drawing as you would have in any other `QGraphicsScene` drawing. To use a pen to plot a line, you simply create a new `QPen` instance and pass it into the `plot` method.

Below we create a `QPen` object, passing in a 3-tuple of `int` values specifying an RGB value (of full red). We could also define this by passing 'r', or a `QColor` object. Then we pass this into `plot` with the *pen* parameter.

PYTHON

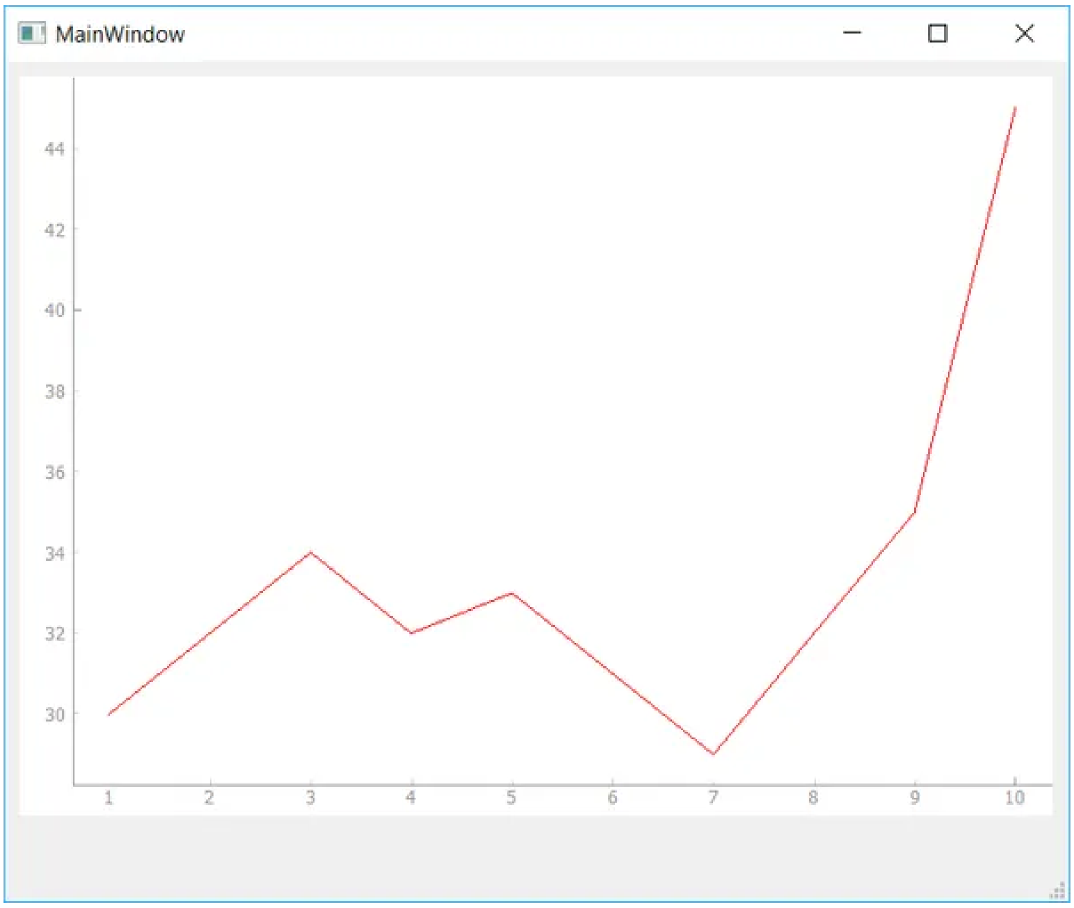
```python
pen = pg.mkPen(color=(255, 0, 0))
self.graphWidget.plot(hour, temperature, pen=pen)
```

The complete code is shown below.

PYTHON

```python
from PyQt5 import QtWidgets
from pyqtgraph import PlotWidget, plot
import pyqtgraph as pg
import sys  # We need sys so that we can pass argv to QApplication
import os
```

```
class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1,2,3,4,5,6,7,8,9,10]
        temperature = [30,32,34,32,33,31,29,32,35,45]

        self.graphWidget.setBackground('w')

        pen = pg.mkPen(color=(255, 0, 0))
        self.graphWidget.plot(hour, temperature, pen=pen)


def main():
    app = QtWidgets.QApplication(sys.argv)
    main = MainWindow()
    main.show()
    sys.exit(app.exec_())


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

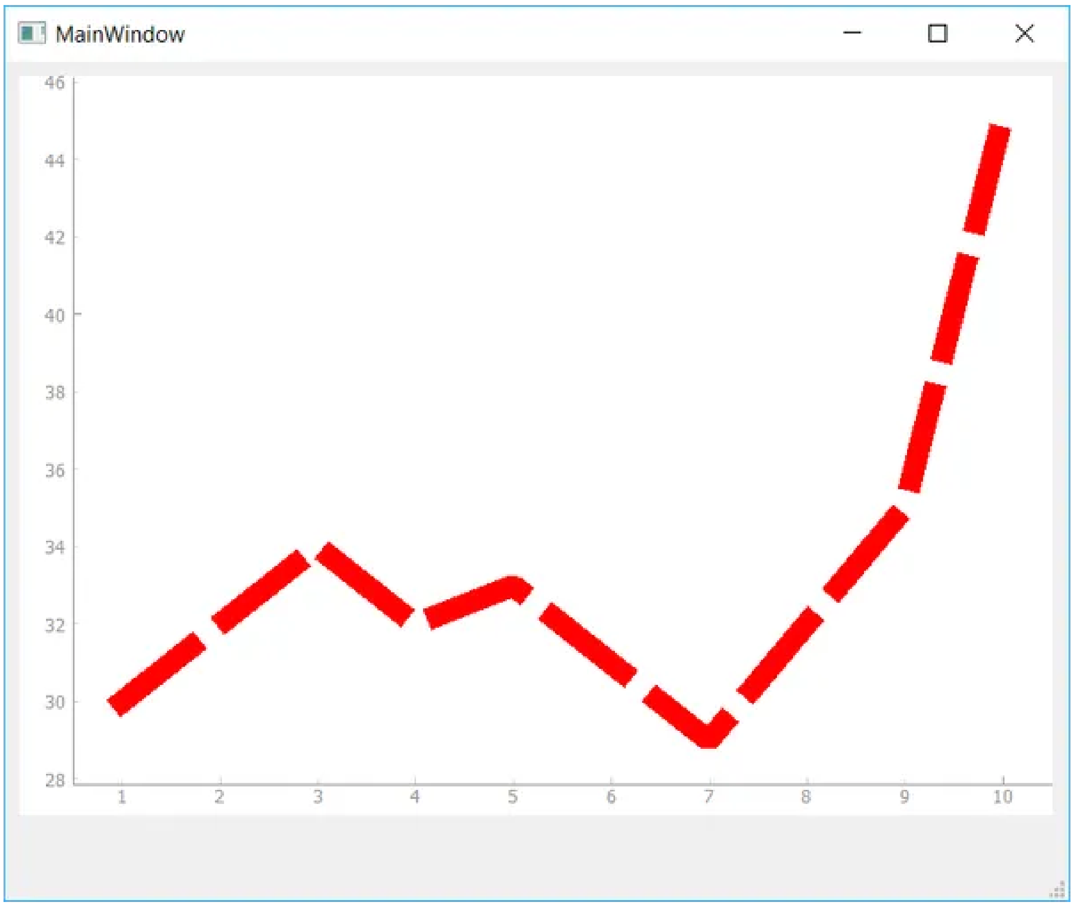


Changing Line Colour

By changing the `QPen` object we can change the appearance of the line, including both line width in pixels and style (dashed, dotted, etc.) using standard Qt line styles. For example, the following example creates a 15px width dashed line in red.

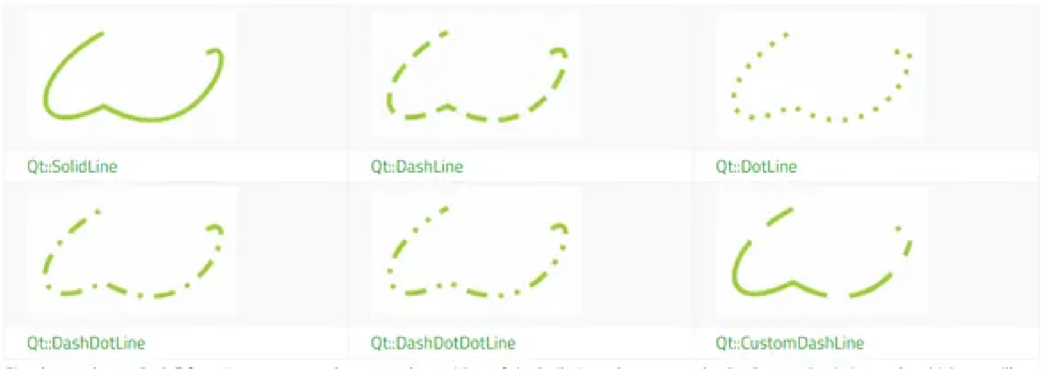
```
pen = pg.mkPen(color=(255, 0, 0), width=15, style=QtCore.Qt.DashLine)
```

The result is shown below, giving a 15px dashed red line.



Changing Line Width and Style

The standard Qt line styles can all be used, including `Qt.SolidLine`, `Qt.DashLine`, `Qt.DotLine`, `Qt.DashDotLine` and `Qt.DashDotDotLine`. Examples of each of these lines are shown in the image below, and you can read more in the Qt Documentation.



Qt Line Types

Line Markers

For many plots it can be helpful to place markers in addition or instead of lines on the plot. To draw a marker on the plot, pass the symbol to use as a marker when calling `.plot` as shown below.

PYTHON

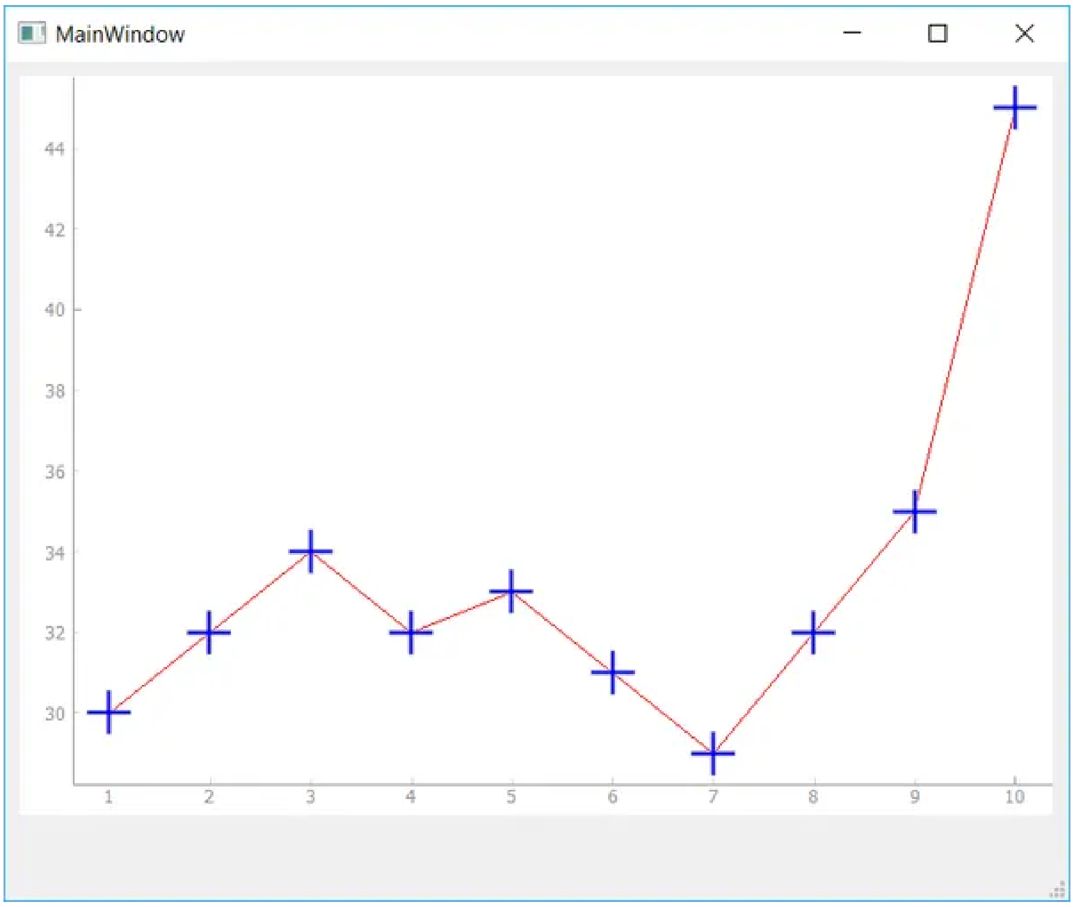
```python
self.graphWidget.plot(hour, temperature, symbol='+')
```

In addition to `symbol` you can also pass in `symbolSize`, `symbolBrush` and `symbolPen` parameters. The value passed as `symbolBrush` can be any colour, or `QBrush` type, while `symbolPen` can be passed any colour or a `QPen` instance. The *pen* is used to draw the outline of the shape, while *brush* is used for the fill.

For example the below code will give a blue cross marker of size 30, on a thick red line.

PYTHON

```python
pen = pg.mkPen(color=(255, 0, 0), width=15, style=QtCore.Qt.DashLine)
self.graphWidget.plot(hour, temperature, pen=pen, symbol='+', symbolSize=30, symbolBrush=('b'))
```



Adding Symbols on Line

In addition to the `+` plot marker, PyQtGraph supports the following standard markers shown in the table below. These can all be used in the same way.

If you have more complex requirements you can also pass in any `QPainterPath` object, allowing you to draw completely custom marker shapes.

Plot Titles

Chart titles are important to provide context to what is shown on a given chart. In PyQtGraph you can add a main plot title using the `setTitle()` method on the `PlotWidget`, passing in your title string.

PYTHON

```
self.graphWidget.setTitle("Your Title Here")
```

You can apply text styles, including colours, font sizes and weights to your titles (and any other labels in PyQtGraph) by passing additional arguments. The available syle arguments are shown below.

The code below sets the color to blue with a font size of 30px.

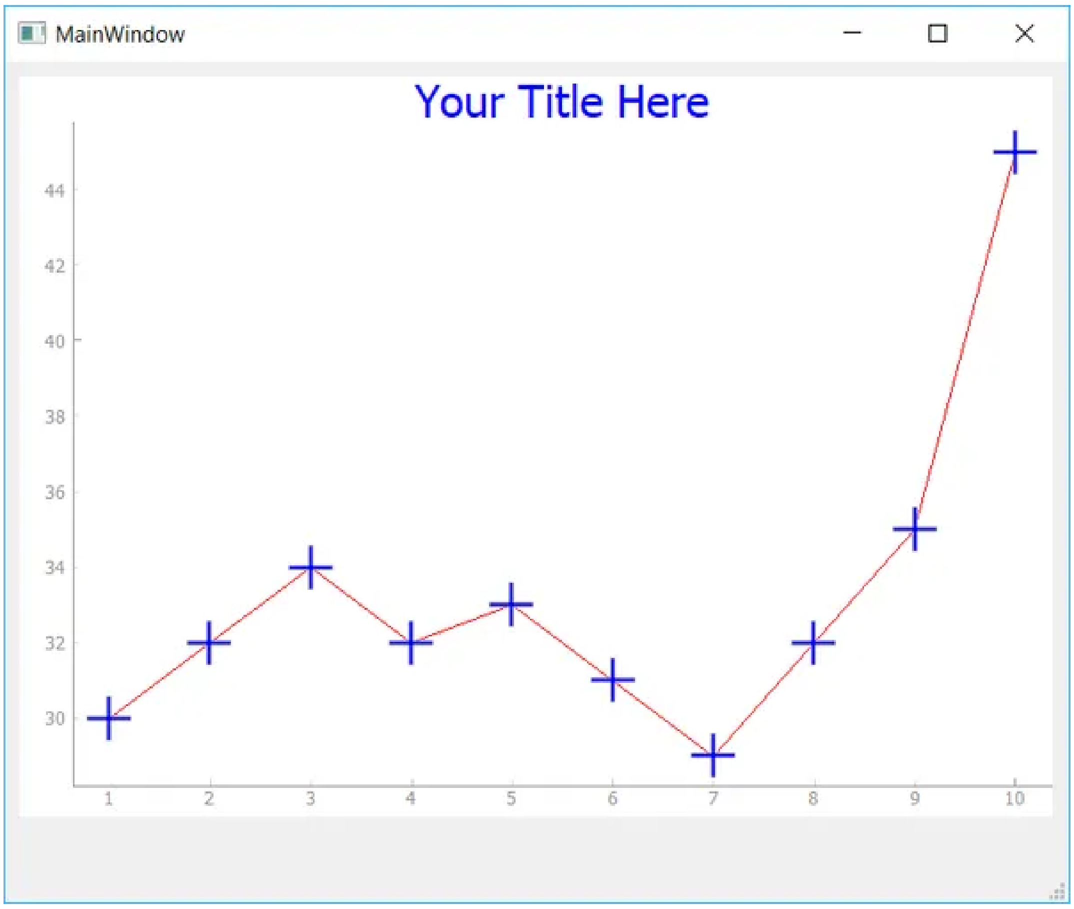
PYTHON

```
self.graphWidget.setTitle("Your Title Here", color="b", size="30pt")
```

You can also style your headers with HTML tag syntax if you prefer, although it's less readable.

PYTHON

```
self.graphWidget.setTitle("<span style=\"color:blue;font-size:30pt\">Your Title Here</span>")
```



Adding Chart Title

Axis Labels

Similar to titles, we can use the `setLabel()` method to create our axis titles. This requires two parameters, *position* and *text*. The *position* can be any one of `'left,'right','top','bottom'` which describe the position of the axis on which the text is placed. The 2nd parameter *text* is the text you want to use for the label.

You can pass additional style parameters into the method. These differ slightly than for the title, in that they need to be valid CSS name-value pairs. For example, the size is now `font-size`. Because the name `font-size` has a hyphen in it, you cannot pass it directly as a parameter, but must use the `**dictionary` method.

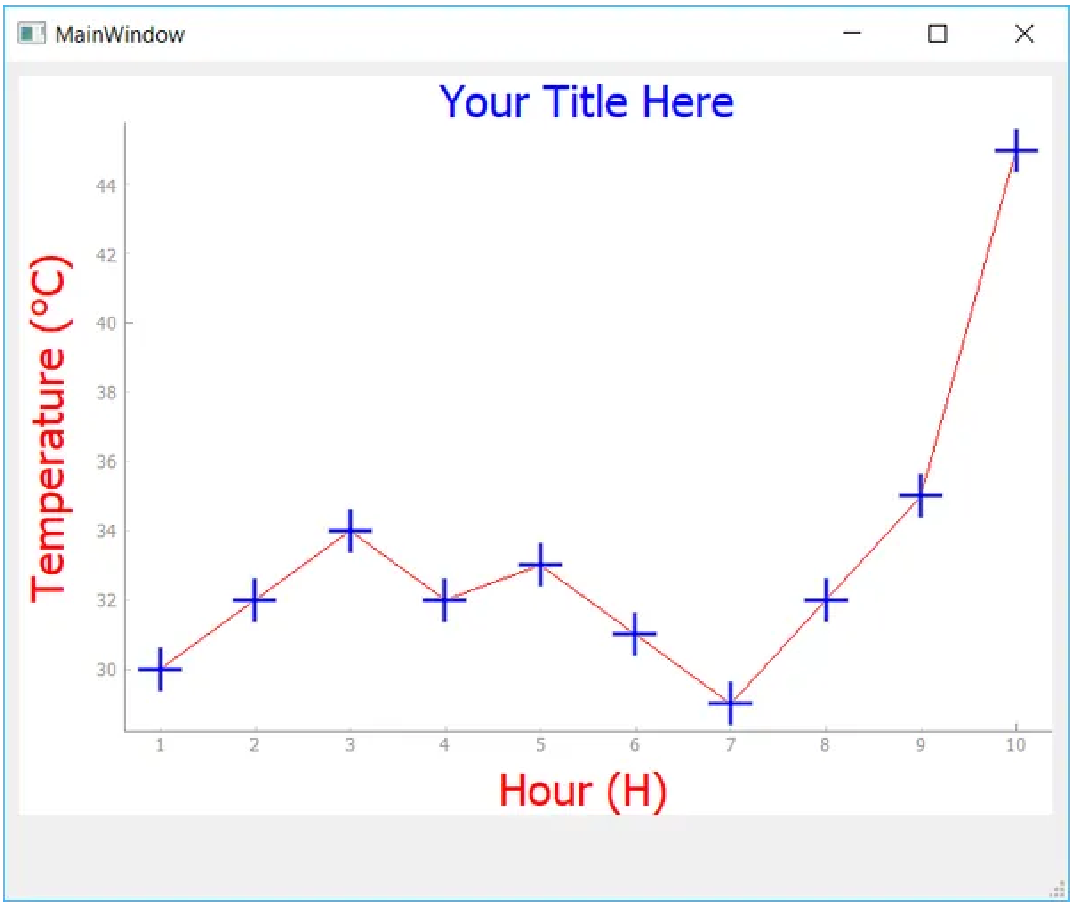
PYTHON

```python
styles = {'color':'r', 'font-size':'20px'}
self.graphWidget.setLabel('left', 'Temperature (°C)', **styles)
self.graphWidget.setLabel('bottom', 'Hour (H)', **styles)
```

These also support HTML syntax for styling if you prefer.

PYTHON

```python
self.graphWidget.setLabel('left', "<span style=\"color:red;font-size:20px\">Temperature (°C)</span>
self.graphWidget.setLabel('bottom', "<span style=\"color:red;font-size:20px\">Hour (H)</span>")
```



Add Axis Labels

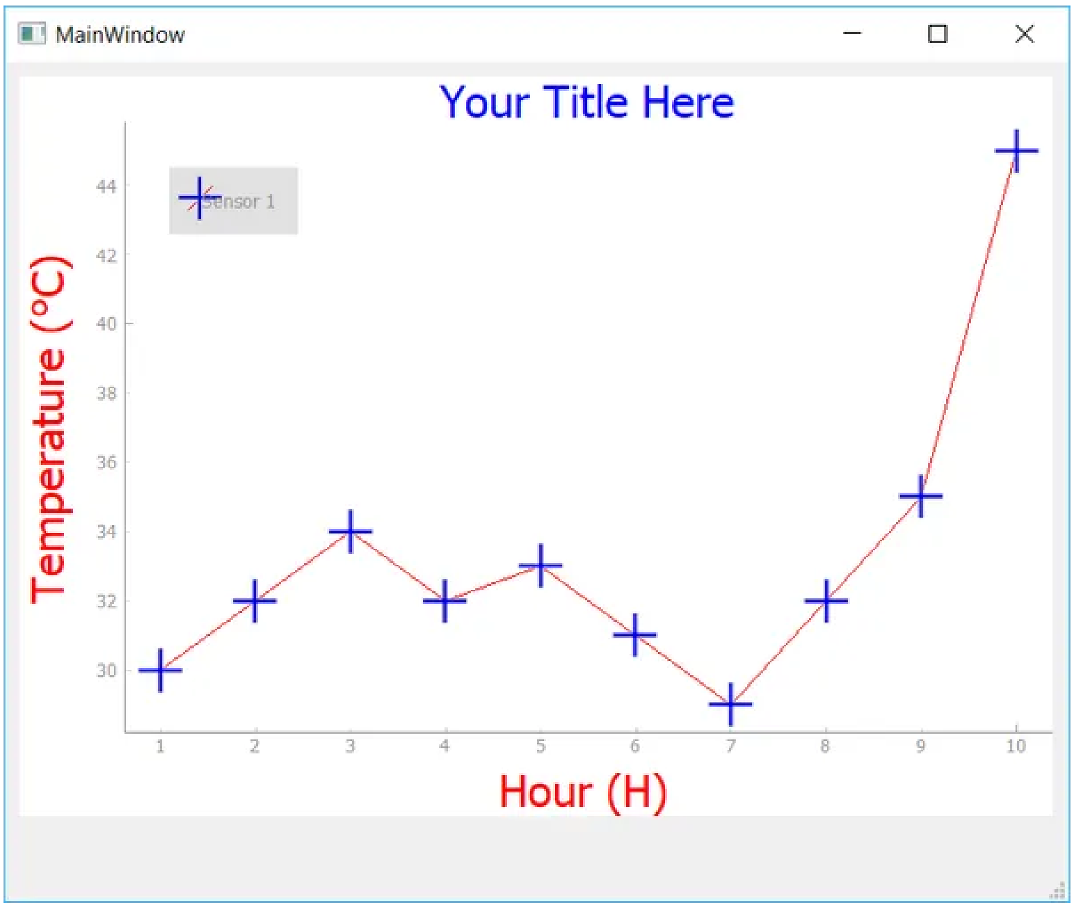
Legends

In addition to the axis and plot titles you will often want to show a legend identifying what a given line represents. This is particularly important when you start adding multiple lines to a plot. Adding a legend to a plot can be accomplished by calling `.addLegend` on the `PlotWidget`, however before this will work you need to provide a name for each line when calling `.plot()`.

The example below assigns a name "Sensor 1" to the line we are plotting with `.plot()`. This name will be used to identify the line in the legend.

PYTHON

```python
self.graphWidget.plot(hour, temperature, name = "Sensor 1",  pen = NewPen, symbol='+', symbolSize=3
self.graphWidget.addLegend()
```



Add Legend

The legend appears in the top left by default. If you would like to move it, you can easily drag and drop the legend elsewhere. You can also specify a default offset by passing a 2-tuple to the `offset` parameter when creating the legend.

Background Grid

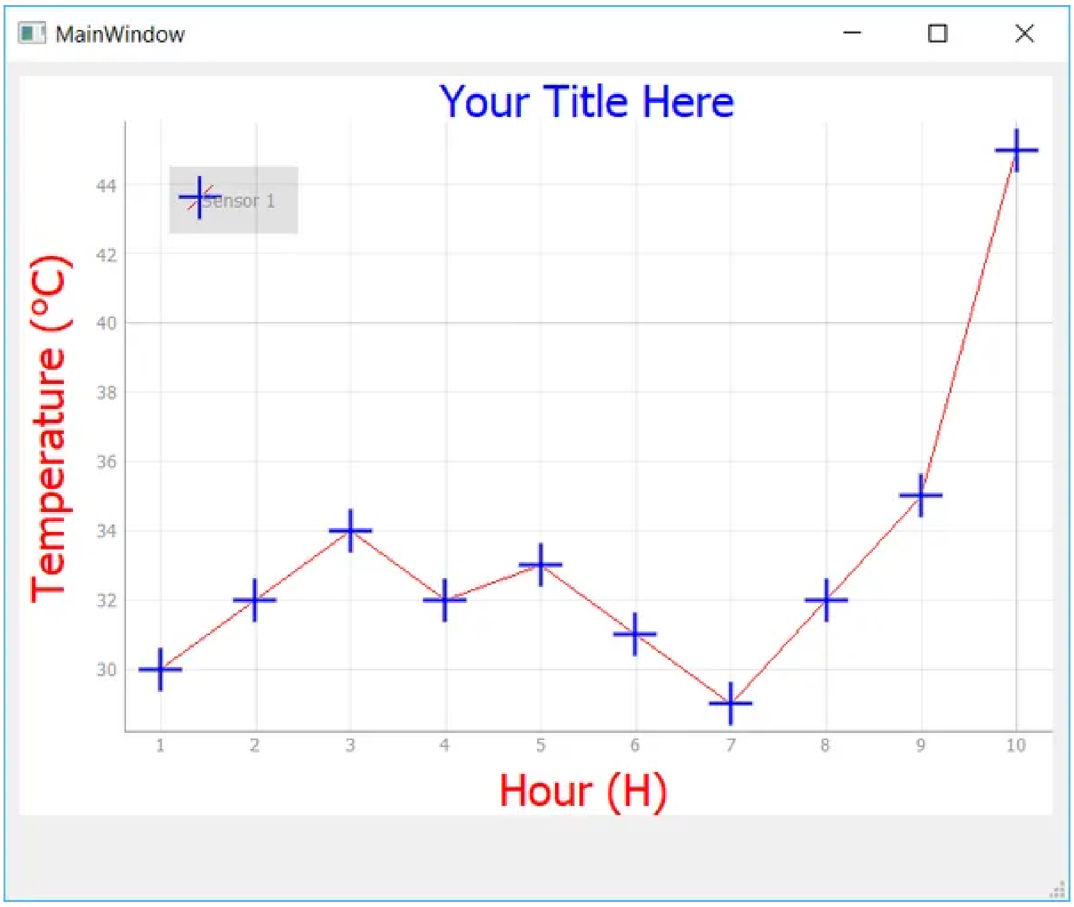
Adding a background grid can make your plots easier to read, particularly when trying to compare relative x & y values against each other. You can turn on a background grid for your plot by calling

`.showGrid` on your `PlotWidget`. You can toggle x and y grids independently.

The following with create the grid for both the X and Y axis.

PYTHON

```
self.graphWidget.showGrid(x=True, y=True)
```



Add Grid

Setting Axis Limits

Sometimes it can be useful to restrict the range of data which is visible on the plot, or to lock the axis to a consistent range regardless of the data input (e.g. a known min-max range). In PyQtGraph this can be done using the `.setXRange()` and `.setYRange()` methods. These force the plot to only show data within the specified ranges on each axis.

Below we set two ranges, one on each axis. The 1st argument is the minimum value and the 2nd is the maximum.

PYTHON

```
self.graphWidget.setXRange(5, 20, padding=0)
self.graphWidget.setYRange(30, 40, padding=0)
```

A optional padding argument causes the range to be set larger than specified by the specified fraction (this between 0.02 and 0.1 by default, depending on the size of the ViewBox). If you want to remove this padding entirely, pass 0.

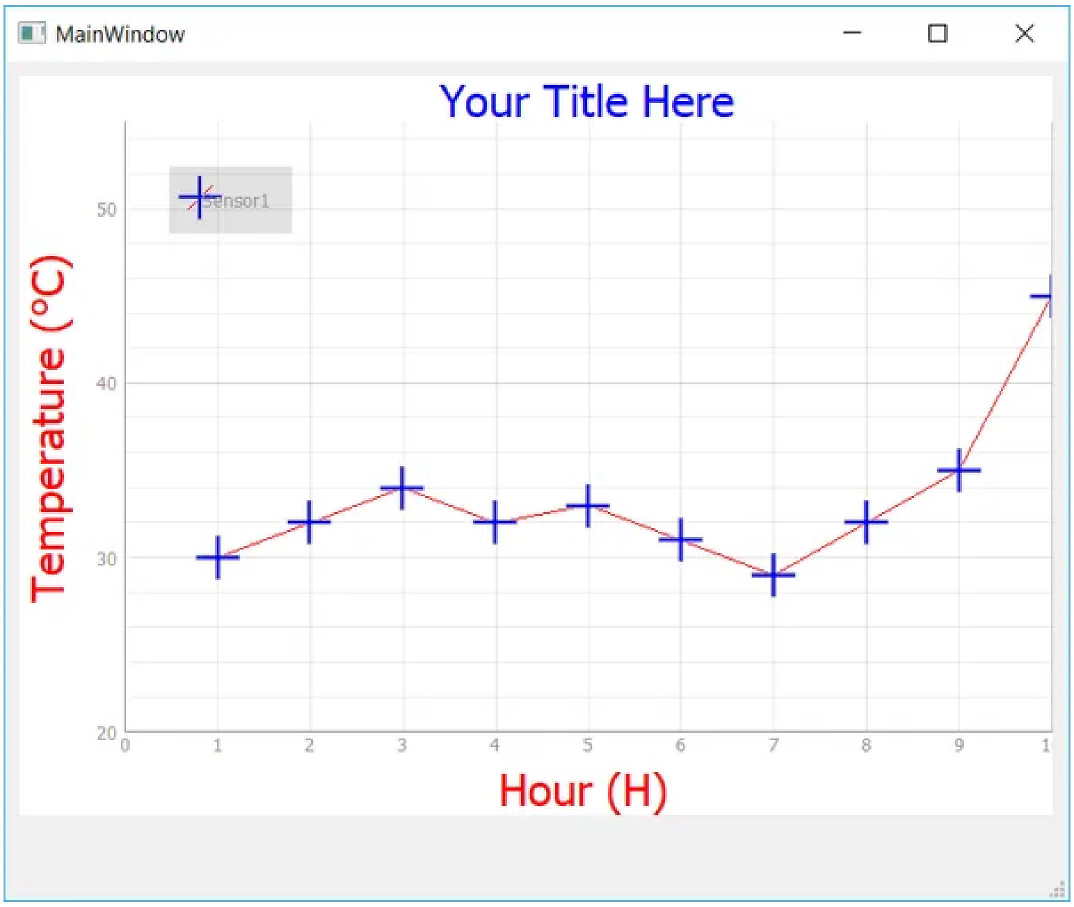
PYTHON

```
self.graphWidget.setXRange(5, 20, padding=0)
self.graphWidget.setYRange(30, 40, padding=0)
```

The complete code so far is shown below:

PYTHON

```
from PyQt5 import QtWidgets, QtCore
from pyqtgraph import PlotWidget, plot
import pyqtgraph as pg
import sys  # We need sys so that we can pass argv to QApplication
import os

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1,2,3,4,5,6,7,8,9,10]
        temperature = [30,32,34,32,33,31,29,32,35,45]

        #Add Background colour to white
        self.graphWidget.setBackground('w')
        # Add Title
        self.graphWidget.setTitle("Your Title Here", color="b", size="30pt")
        # Add Axis Labels
        styles = {"color": "#f00", "font-size": "20px"}
        self.graphWidget.setLabel("left", "Temperature (°C)", **styles)
        self.graphWidget.setLabel("bottom", "Hour (H)", **styles)
        #Add legend
        self.graphWidget.addLegend()
        #Add grid
        self.graphWidget.showGrid(x=True, y=True)
        #Set Range
        self.graphWidget.setXRange(0, 10, padding=0)
        self.graphWidget.setYRange(20, 55, padding=0)

        pen = pg.mkPen(color=(255, 0, 0))
        self.graphWidget.plot(hour, temperature, name="Sensor 1",  pen=pen, symbol='+', symbolSize=

def main():
    app = QtWidgets.QApplication(sys.argv)
    main = MainWindow()
    main.show()
    sys.exit(app.exec_())

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



Set Axis Range

Plotting multiple lines

It is common for plots to involve more than one line. In PyQtGraph this is as simple as calling `.plot()` multiple times on the same `PlotWidget`. In the following example we're going to plot two lines of similar data, using the same line styles, thicknesses etc. for each, but changing the line colour.

To simplify this we can create our own custom `plot` method on our `MainWindow`. This accepts `x` and `y` parameters to plot, the name of the line (for the legend) and a colour. We use the colour for both the line and marker colour.

PYTHON

```python
    def plot(self, x, y, plotname, color):
        pen = pg.mkPen(color=color)
        self.graphWidget.plot(x, y, name=plotname, pen=pen, symbol='+', symbolSize=30, symbolBrush=
```

To plot separate lines we'll create a new array called `temperature_2` and populate it with random numbers similar to `temperature` (now `temperature_1`). Plotting these alongside each other allows us to compare them together.

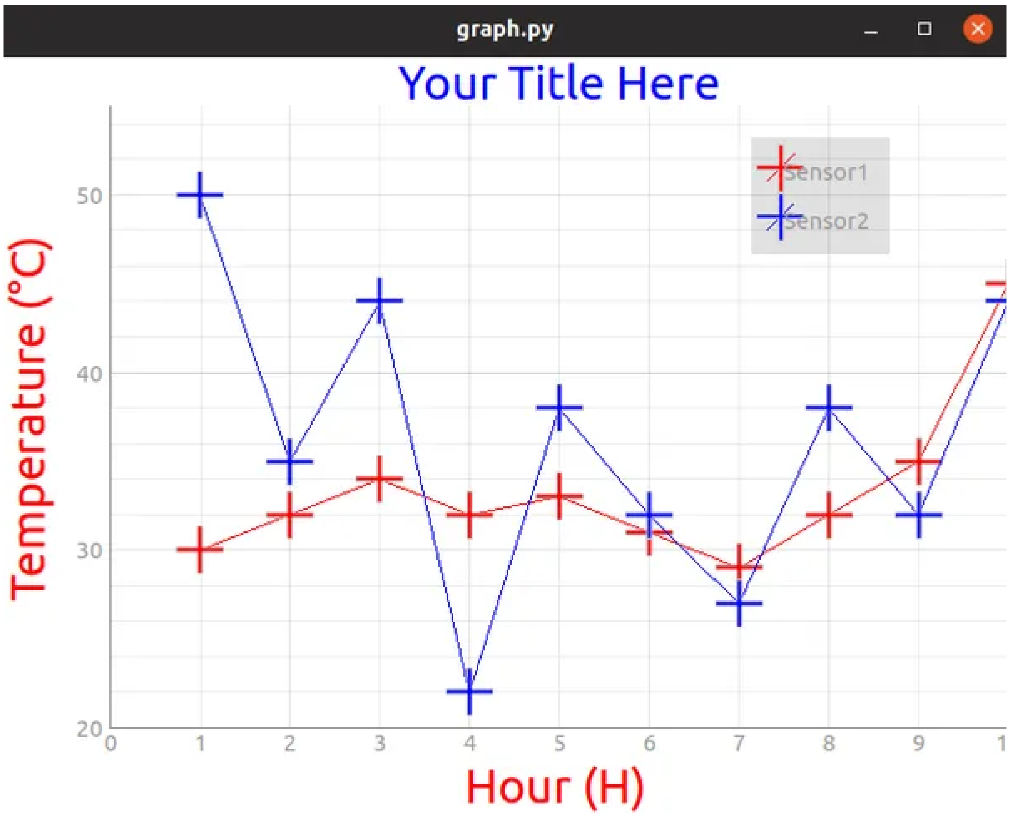
Now, you can call plot function twice and this will generate 2 lines on the plot.

PYTHON

```python
self.plot(hour, temperature_1, "Sensor1", 'r')
self.plot(hour, temperature_2, "Sensor2", 'b')
```

PYTHON

```python
from PyQt5 import QtWidgets, QtCore
from pyqtgraph import PlotWidget, plot
import pyqtgraph as pg
import sys  # We need sys so that we can pass argv to QApplication
import os

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        hour = [1,2,3,4,5,6,7,8,9,10]
        temperature_1 = [30,32,34,32,33,31,29,32,35,45]
        temperature_2 = [50,35,44,22,38,32,27,38,32,44]

        #Add Background colour to white
        self.graphWidget.setBackground('w')
        # Add Title
        self.graphWidget.setTitle("Your Title Here", color="b", size="30pt")
        # Add Axis Labels
        styles = {"color": "#f00", "font-size": "20px"}
        self.graphWidget.setLabel("left", "Temperature (°C)", **styles)
        self.graphWidget.setLabel("bottom", "Hour (H)", **styles)
        #Add legend
        self.graphWidget.addLegend()
        #Add grid
        self.graphWidget.showGrid(x=True, y=True)
        #Set Range
        self.graphWidget.setXRange(0, 10, padding=0)
        self.graphWidget.setYRange(20, 55, padding=0)

        self.plot(hour, temperature_1, "Sensor1", 'r')
        self.plot(hour, temperature_2, "Sensor2", 'b')

    def plot(self, x, y, plotname, color):
        pen = pg.mkPen(color=color)
        self.graphWidget.plot(x, y, name=plotname, pen=pen, symbol='+', symbolSize=30, symbolBrush=

def main():
    app = QtWidgets.QApplication(sys.argv)
    main = MainWindow()
    main.show()
    sys.exit(app.exec_())

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



2 Line Graph

Play around with this function, customising your markers, line widths, colours and other parameters.

Clearing the plot

Finally, sometimes you might want to clear and refresh the plot periodically. You can easily do that by calling `.clear()`.

```python
self.graphWidget.clear()
```

This will remove the lines from the plot but keep all other attributes the same.

Updating the plot

While you *can* simply clear the plot and redraw all your elements again, this means Qt has to destroy and recreate all your `QGraphicsScene` objects. For small or simple plots this is probably not noticeable, but if you want to create high-peformance streaming plots it is much better to update the data in place. PyQtGraph takes the new data and updates the plotted line to match without affecting any other elements in the plot.

To update a line we need a reference to the line object. This reference is returned when first creating the line using `.plot` and we can simply store this in a variable. Note that this is a reference to the *line* not to the plot.

PYTHON

```python
my_line_ref = graphWidget.plot(x, y)
```

Once we have the reference, updating the plot is simply a case of calling `.setData` on the reference to apply the new data. In the example below we've take our simple plot demo and expanded it to take a reference to the line.

PYTHON

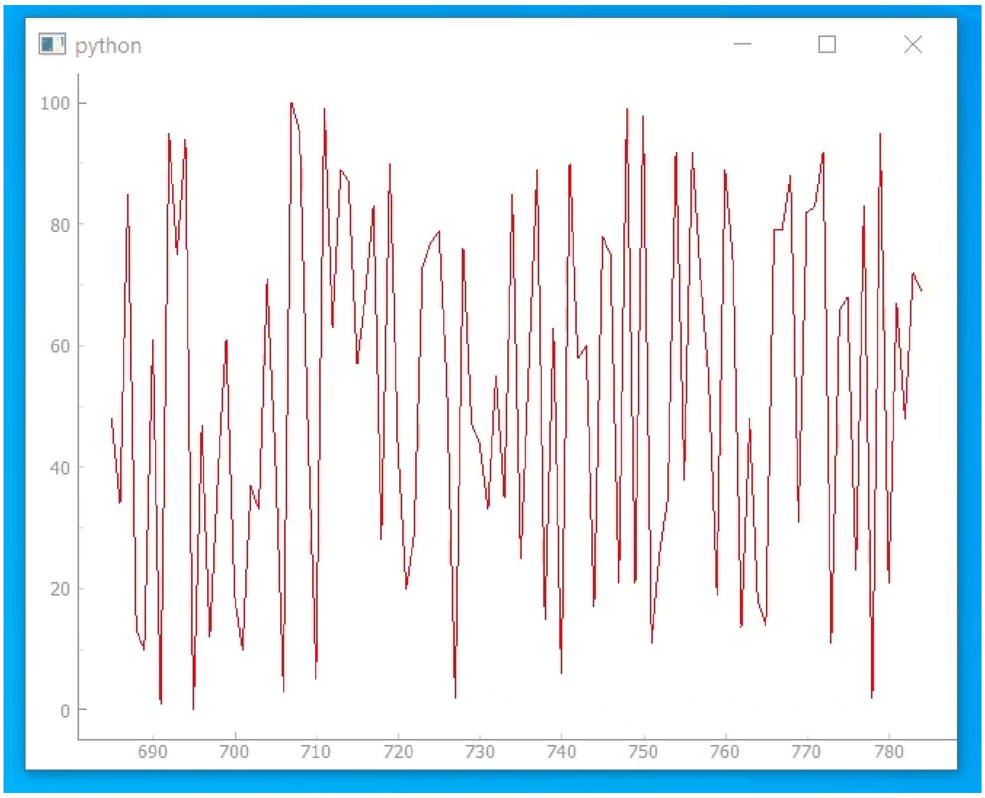
```python
from PyQt5 import QtWidgets, QtCore
from pyqtgraph import PlotWidget, plot
import pyqtgraph as pg
import sys  # We need sys so that we can pass argv to QApplication
import os
from random import randint

class MainWindow(QtWidgets.QMainWindow):

    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        self.graphWidget = pg.PlotWidget()
        self.setCentralWidget(self.graphWidget)

        self.x = list(range(100))  # 100 time points
        self.y = [randint(0,100) for _ in range(100)]  # 100 data points

        self.graphWidget.setBackground('w')

        pen = pg.mkPen(color=(255, 0, 0))
        self.data_line =  self.graphWidget.plot(self.x, self.y, pen=pen)


app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
w.show()
sys.exit(app.exec_())
```

We're going to update our data every 50ms, although PyQtGraph can plot data much more quickly than this it can get hard to watch! To do this we define a Qt timer, and set it to call a custom method `update_plot_data` where we'll change the data. We define this timer in the `__init__` block so it is automatically started. Add the following to the window class:

PYTHON

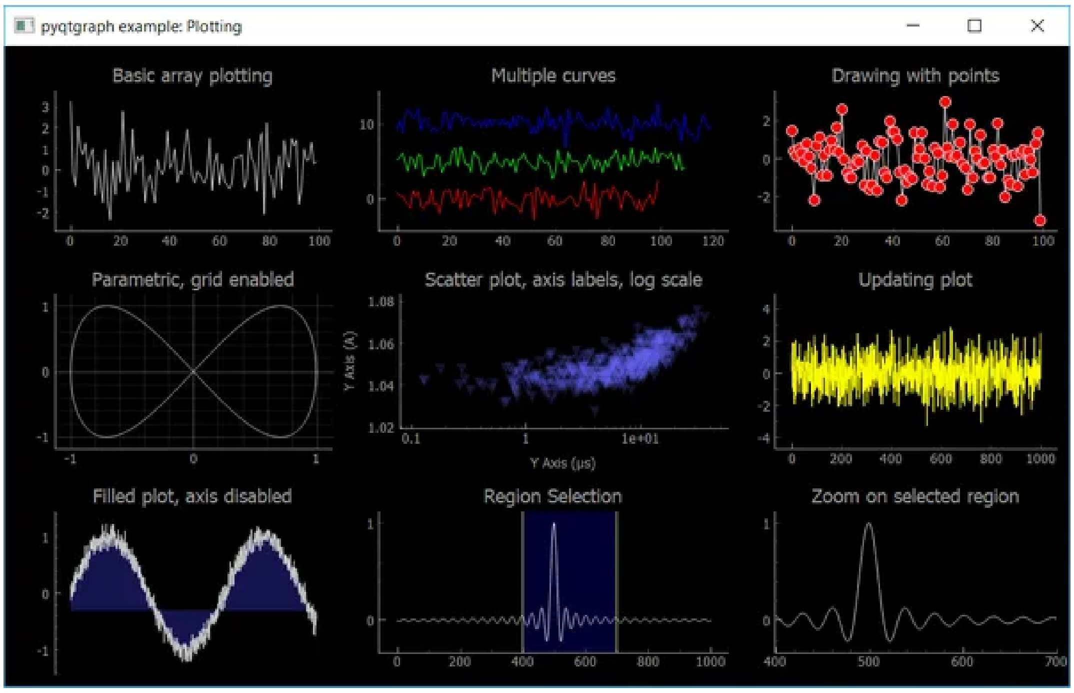
```python
        # ... init continued ...
        self.timer = QtCore.QTimer()
        self.timer.setInterval(50)
        self.timer.timeout.connect(self.update_plot_data)
        self.timer.start()

    def update_plot_data(self):

        self.x = self.x[1:]  # Remove the first y element.
        self.x.append(self.x[-1] + 1)  # Add a new value 1 higher than the last.

        self.y = self.y[1:]  # Remove the first
        self.y.append( randint(0,100))  # Add a new random value.

        self.data_line.setData(self.x, self.y)  # Update the data.
```

If you run the app you will see a plot with random data scrolling rapidly to the left, with the X values also updating and scrolling in time, as if streaming data. You can replace the random data with your own real data, taken for example from a live sensor readout or API. PyQtGraph is performant enough to support multiple simultaneous plots using this method.



Conclusion

In this tutorial we've discovered how to draw simple plots with PyQtGraph and customize lines, markers and labels. For a complete overview of PyQtGraph methods and capabilities see the PyQtGraph Documentation & API Reference. The PyQtGraph repository on Github also has complete set of more complex example plots in Plotting.py (shown below).



PyQtGraph Repo Example (Plotting.py)

Next

Plotting with PyQtGraph was written by John Lim .

John is a developer from Kuala Lumpur, Malaysia who works as a Senior R&D Engineer.

Interested in contributing to the site? Find out more.

1/2

Mark As Complete

Contents

Getting started
Creating a PyQtGraph widget
Styling plots
Plot Titles
Axis Labels
Legends
Background Grid
Setting Axis Limits
Plotting multiple lines
Clearing the plot
Updating the plot
Conclusion

Share

Twitter

Facebook

Reddit

Plotting with PyQtGraph was published in tutorials on October 12, 2019 (updated August 15, 2021) and tagged COLOUR QT STYLE LINE PYQTGRAPH MARKER PLOT PYQT PYTHON

Continue reading →

Q&A: How can I enable editing on a QTableView? PYQT

In the model views course we covered Displaying tabular data in Qt5 ModelViews. This takes a data source, for example a list of list objects, a numpy array or a Pandas DataTable and displays it in a Qt table view. But often, displaying is just the first step -- you also ... More

Topics

PyQt
PySide
NumPy
Pandas & Datatables
GUI
Plotting in GUIs
Packaging GUI applications

Installation
First steps with PyQt5
Example PyQt5 Apps
Widget Library
Reusable code & snippets

PyQt5 Book / PySide2 Book
PyQt5 tutorial / PySide2 tutorial
Latest articles
Frequently Asked Questions
Legacy account login

About
Hire me
1-on-1 mentoring
Write with me
Contact me
Licensing, Privacy & Legal