

数据可视化

Python 的主要优势之一在于数据科学和可视化，它使用 Pandas、numpy 和 sklearn 等工具进行数据分析。使用 PyQt6 构建图形用户界面应用程序，您可以直接从应用程序中访问所有这些 Python 工具，从而构建复杂的数据驱动型应用程序和交互式仪表板。我们已经介绍了模型视图，它允许我们以列表和表格的形式显示数据。在本章中，我们将探讨这个难题的最后一块拼图——可视化数据。

在使用 PyQt6 开发应用程序时，您有两个主要选择——matplotlib（它也提供对 Pandas 图表的访问权限）和 PyQtGraph，后者使用 Qt native 图形创建图表。在本章中，我们将探讨如何利用这些库在您的应用程序中可视化数据。

29. 使用 PyQtGraph 进行数据可视化

虽然您可以在 PyQt6 中嵌入 `matplotlib` 图表，但使用体验并不完全原生。对于简单且高度交互的绘图，您可能需要考虑使用 PyQtGraph 代替。PyQtGraph 基于 PyQt6 本机 `QGraphicsScene` 构建，可以提供更好的绘图性能，特别是对于实时数据，同时提供交互性，并能够使用 Qt 图形控件轻松自定义绘图。

在本章中，我们将介绍使用 PyQtGraph 创建绘图控件的第一步，然后演示如何使用线条颜色、线条类型、轴标签、背景颜色和绘制多条线条来定制绘图。

开始使用

要使用 PyQtGraph 与 PyQt6，您首先需要将该包安装到您的 Python 环境中。您可以使用 `pip` 进行安装。

撰写本文时，PyQt6 还非常新，因此您需要使用 PyQtGraph 的开发者安装版本。

```
pip install git+https://github.com/pyqtgraph/pyqtgraph@master
```

安装完成后，您应该能够像往常一样导入该模块。

创建 PyQtGraph 控件

在 PyQtGraph 中，所有图都是使用 `PlotWidget` 控件创建的。该控件提供了一个包含的画布，可以在上面添加和配置任何类型的图。在后台，该图控件使用 Qt 本地的 `QGraphicsScene`，这意味着它快速、高效且易于与应用程序的其他部分集成。您可以像创建其他控件一样创建 `PlotWidget`。

以下是基本模板应用程序的示例，该应用程序在一个 `QMainWindow` 中包含一个 `PlotWidget`。



在以下示例中，我们将创建 PyQtGraph 控件。但是，您也可以从 Qt Designer 中嵌入 PyQtGraph 控件。

Listing 220. plotting/pyqtgraph_1.py

```
import sys
```

```

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph

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

        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]

        # 绘制数据: x、y 值
        self.graphwidget.plot(hour, temperature)

app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()

```



在以下所有示例中，我们使用 `import pyqtgraph as pg` 导入 PyQtGraph。这是 PyQtGraph 示例中常见的约定，旨在保持代码整洁并减少重复输入。如果您更喜欢，也可以使用 `import pyqtgraph` 进行导入。

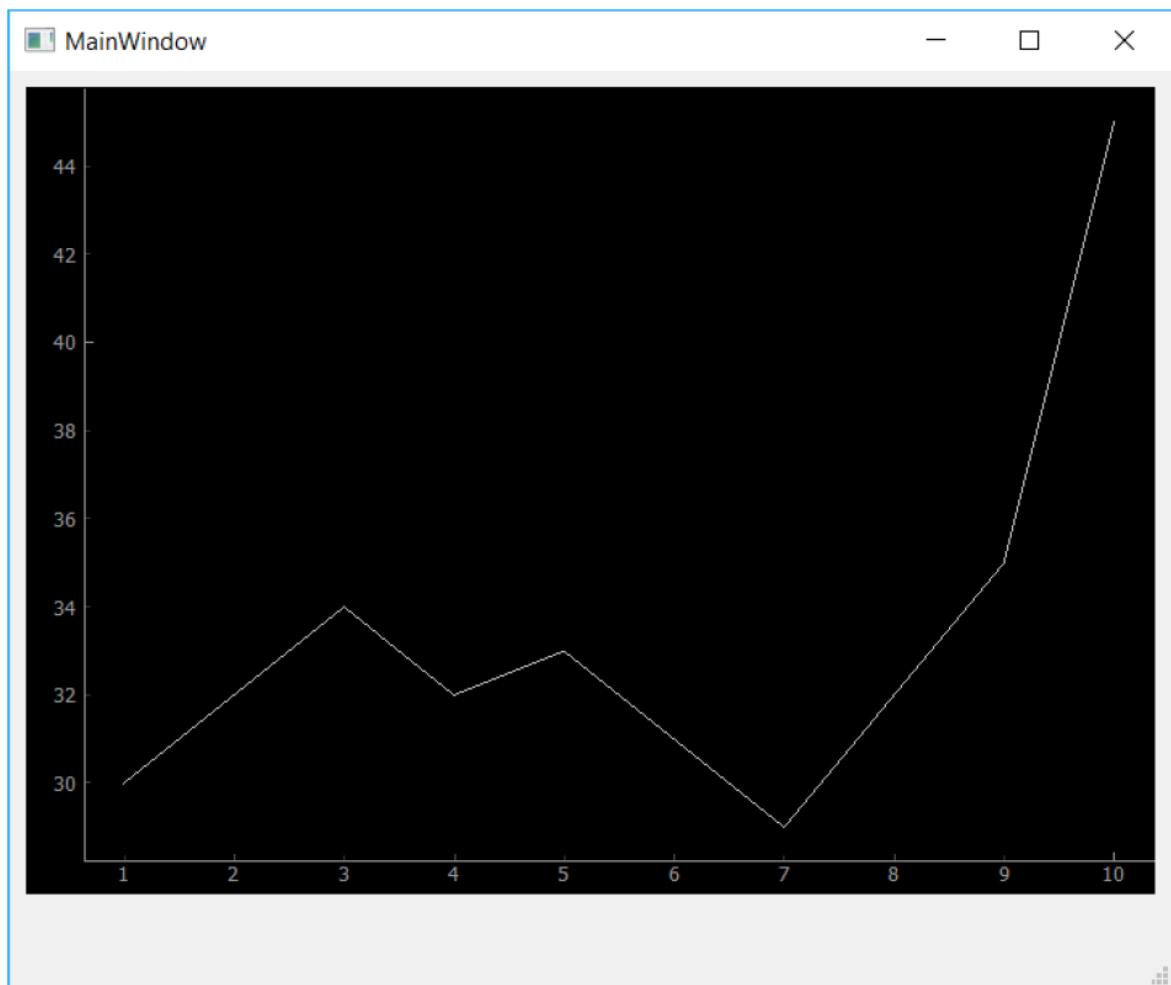


图216：显示虚拟数据的自定义 PyQtGraph 控件。

PyQtGraph 的默认绘图样式非常简单——黑色背景，一条细细的（几乎看不见的）白色线条。在下一节中，我们将看看 PyQtGraph 中有哪些可用的选项，以改善绘图的外观和可用性。

样式绘制

PyQtGraph 使用 Qt 的 `QGraphicsScene` 来绘制图形。这使我们能够访问所有标准的 Qt 线条和形状样式选项，以用于绘图。然而，PyQtGraph 提供了一个 API，用于绘图和管理图画布。

下面我们将介绍创建和自定义自己的图所需的最常见的样式功能。

背景颜色

从上面的应用程序骨架开始，我们可以更改背景颜色，方法是调用 `Plotwidget` 实例（在 `self.graphwidget` 中）的 `.setBackground` 方法。下面的代码将背景设置为白色，方法是传入字符串 `'w'`。

```
self.graphwidget.setBackground('w')
```

您可以随时设置（并更新）图表的背景颜色。

Listing 221. plotting/pyqtgraph_2.py

```
import sys

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph
```

```

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

        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)

app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()

```

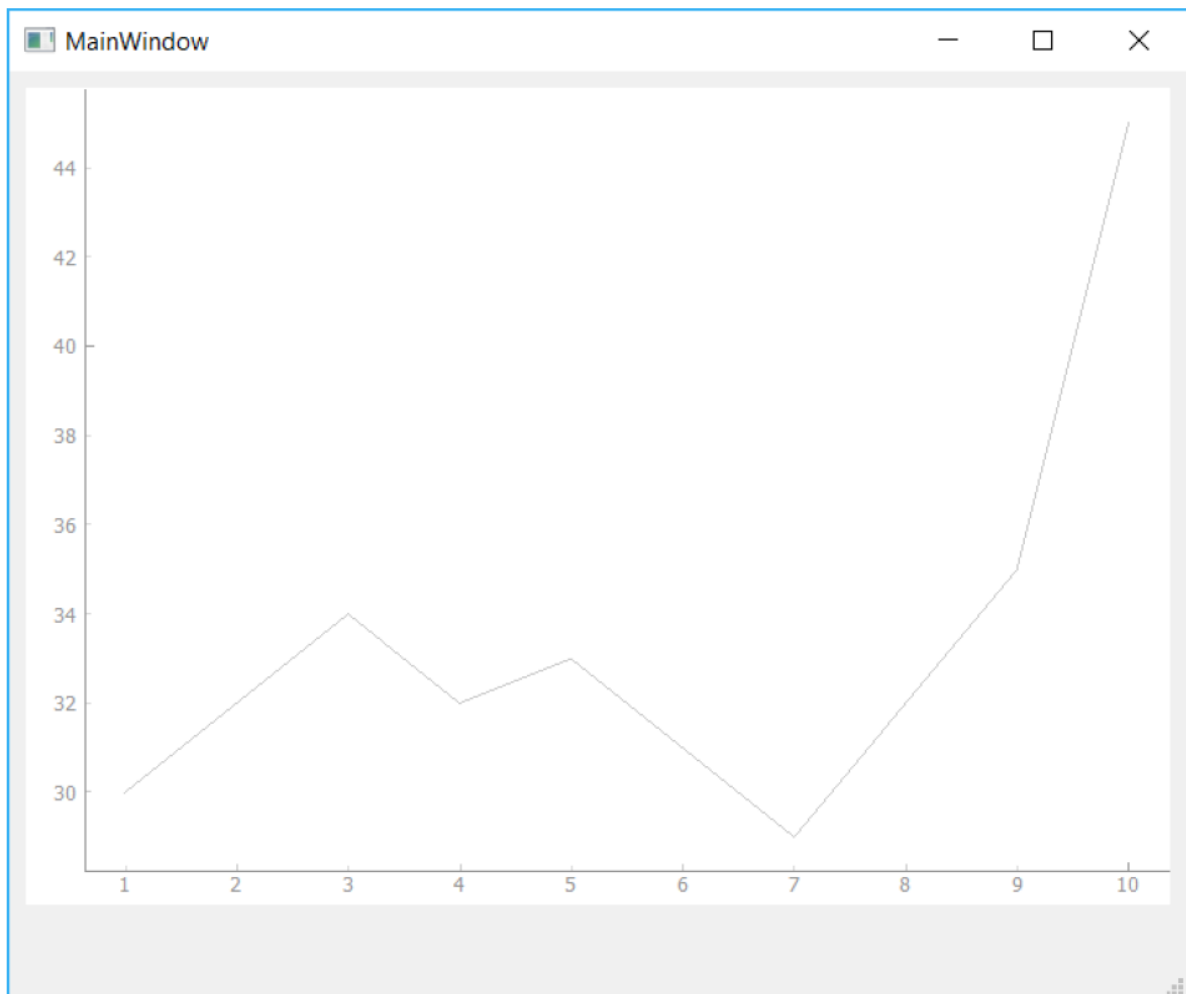


图217：将PyQtGraph的绘图背景改为白色

使用单个字母可以生成多种简单颜色，这些颜色基于 `Matplotlib` 中使用的标准颜色。它们大多较为常见，唯一例外的是'k'用于表示黑色。

Table 7. Common color codes

颜色	字母代号
blue	b
green	g
red	r
cyan (明亮的蓝绿色)	c
magenta (亮粉色)	m
yellow	y
black	k
white	w

除了这些单字母代码外，您还可以使用十六进制表示法设置颜色例如，使用字符串 `#672922`。

```
self.graphwidget.setBackground('#bbccaa') # 十六进制
```

RGB 和 RGBA 值可以分别作为 3 元组或 4 元组传递，使用 0-255 的值。

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

最后，您还可以直接使用Qt的 `QColor` 类型来指定颜色。

```
self.graphwidget.setBackground(QtGui.QColor(100,50,254,25))
```

如果您在应用程序的其他地方使用特定的 `QColor` 对象，或者将图形的背景设置为图形用户界面的默认背景颜色，此功能非常有用。

```
color = self.palette().color(QtGui.QPalette.window) # 获取默认窗口背景，
self.graphwidget.setBackground(color)
```

线条颜色、宽度和样式

PyQtGraph 中的线条使用标准的 Qt `QPen` 类型绘制。这使您能够像在任何其他 `QGraphicsScene` 绘图中一样，对线条绘制拥有完全的控制权。要使用笔来绘制线条，您只需创建一个新的 `QPen` 实例，并将其传递给 `plot` 方法。

下面我们创建一个 `QPen` 对象，传入一个包含三个整数值元组，指定一个RGB 值（全红）。我们也可以通过传入 `'r'` 或一个 `QColor` 对象来定义它。然后将此对象传入 `plot` 函数的 `pen` 参数中。

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

完整的代码如下所示：

```
import sys
```

```

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph

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

        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)

app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()

```



图218：更改线条颜色

通过更改 `QPen` 对象，我们可以更改线条的外观，包括使用标准 Qt 线条样式更改线条的像素宽度和样式（虚线、点线等）。例如，以下示例创建了一条 15 像素宽的红色虚线。

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

结果如下所示，显示一条15像素的红色虚线。

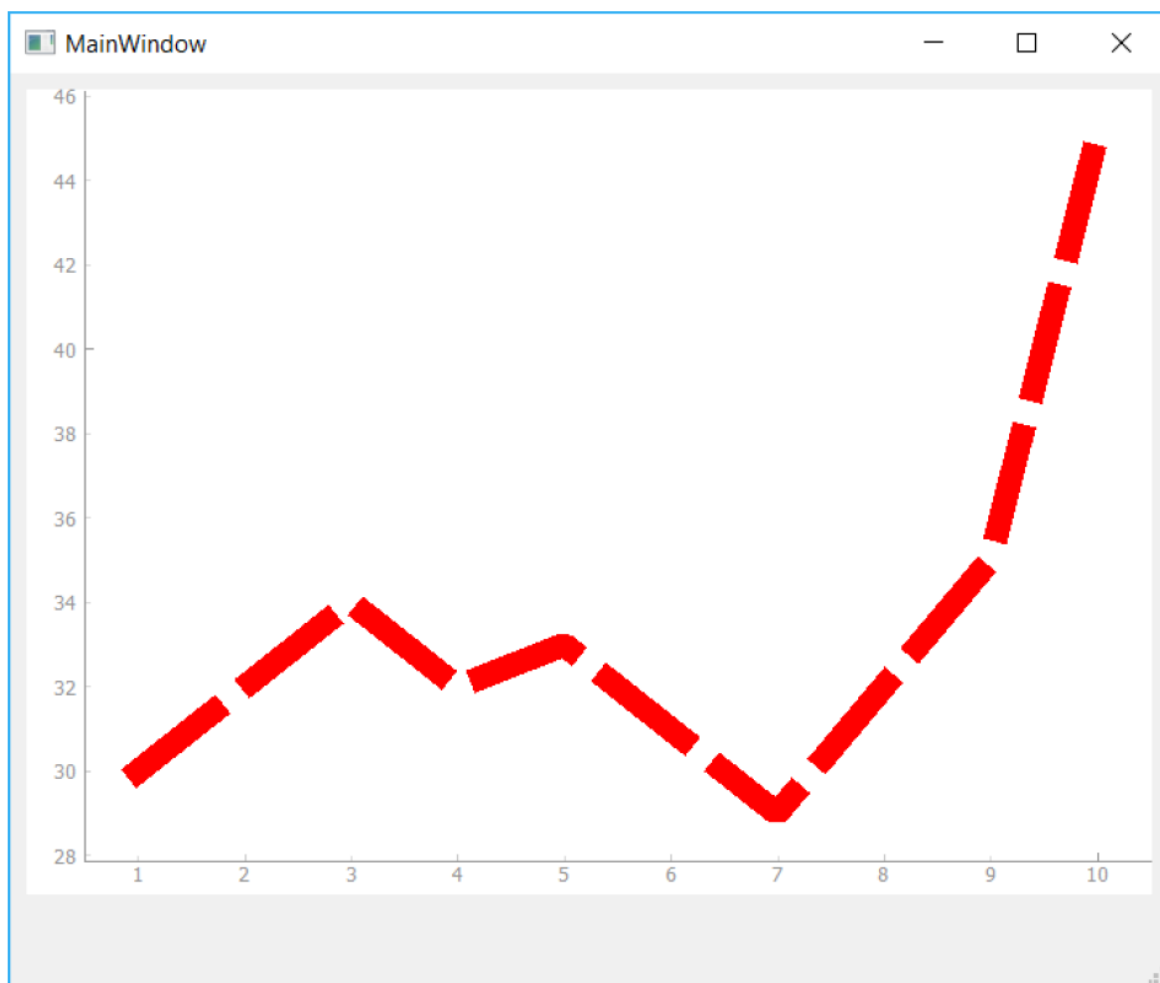


图219：更改线宽和样式。

您可以使用所有标准的 Qt 线条样式，包括 `Qt.PenStyle.SolidLine`、`Qt.PenStyle.DashLine`、`Qt.PenStyle.DotLine`、`Qt.PenStyle.DashDotLine` 和 `Qt.PenStyle.DashDotDotLine`。下图显示了这些线条的示例，您可以在 [Qt 文档](#) 中阅读更多相关内容。

线条标记器

对于许多图表，在图表上添加标记（而非或替代线条）可能会有帮助。要在图表上绘制标记，可在调用 `.plot` 时传入要用作标记的符号，如下所示：

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

除了 `symbol` 外，您还可以传递符号大小（`symbolSize`）、符号画笔（`symbolBrush`）和符号钢笔（`symbolPen`）参数。传递给 `symbolBrush` 的值可以是任何颜色，或 `QBrush` 类型，而符号钢笔可以传递任何颜色或 `QPen` 实例。钢笔用于绘制形状的轮廓，而画笔用于填充。

例如，下面的代码将生成一个大小为30的蓝色十字标记，位于一条粗厚的红色线条上。

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

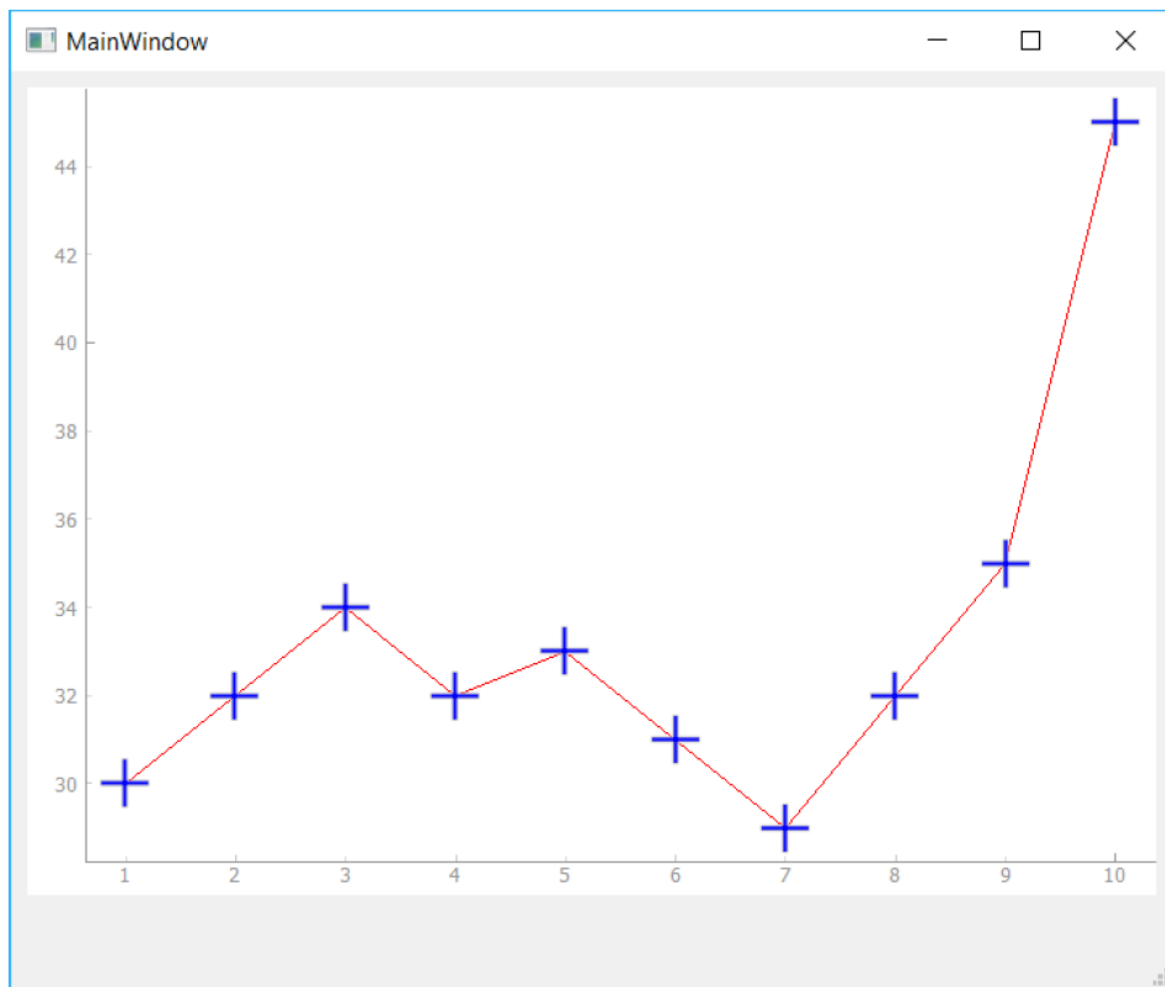


图220：每个数据点上都显示了符号。

除了 **+** 图标标记外，PyQtGraph还支持以下标准标记，如表中所示。这些标记均可按相同方式使用。

变量	标记的类型
o	圆形
s	方形
t	三角形
d	菱形
+	交叉点



如果您有更复杂的需求，还可以传入任何 QPainterPath 对象，从而能够绘制完全自定义的标记形状。

图表标题

图表标题对于提供图表所展示内容的上下文非常重要。在PyQtGraph中，您可以通过调用 `PlotWidget` 上的 `setTitle()` 方法，并传入标题字符串，来添加主图表标题。

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

您可以通过传递附加参数来为标题（以及 PyQtGraph 中的任何其他标签）应用文本样式，包括颜色、字体大小和粗细。可用的样式参数如下所示：

样式	类型
color	(str) e.g. <code>CCFF00</code>
size	(str) e.g. <code>8pt</code>
bold	(bool) <code>True</code> or <code>False</code>
italic	

下面的代码将颜色设置为蓝色，字体大小为30pt。

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

如果您愿意，还可以使用 HTML 标签语法来设置标题样式，尽管这样会降低可读性。

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

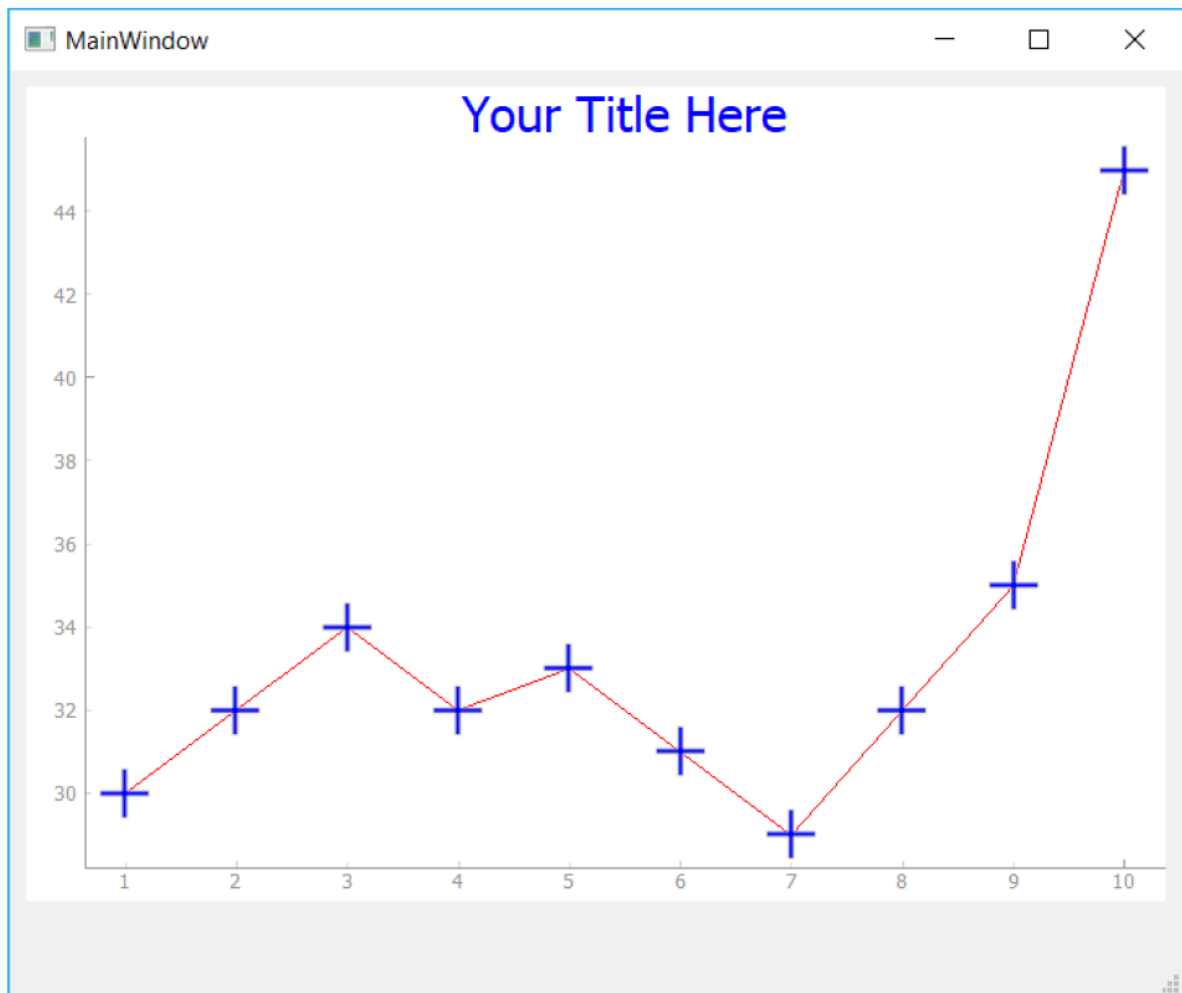


图221：带有样式标题的图例

坐标轴标题

与标题类似，我们可以使用 `setLabel()` 方法来创建轴标题。此方法需要两个参数：位置和文本。位置可以是 `left`、`right`、`top` 或 `bottom` 中的任意一个，用于描述轴上文本的位置。第二个参数文本是您希望用于标签的文本。

您可以将额外的样式参数传递给该方法。这些参数与标题的参数略有不同，因为它们需要是有效的 CSS 名称-值对。例如，大小现在是 `font-size`。由于名称 `font-size` 中包含连字符，因此您不能将其直接作为参数传递，而必须使用 `**dictionary` 方法

```
styles = {'color': 'r', 'font-size': '30pt'}
self.graphwidget.setLabel('left', 'Temperature (°C)', **styles)
self.graphwidget.setLabel('bottom', 'Hour (H)', **styles)
```

这些也支持 HTML 语法，您可以自由选用。

```
self.graphwidget.setLabel('left', "<span  
style=\"color:red;fontsize:30px\">Temperature (°C)</span>")
self.graphwidget.setLabel('bottom', "<span style=\"color:red;fontsize:30px\">Hour  
(H)</span>")
```

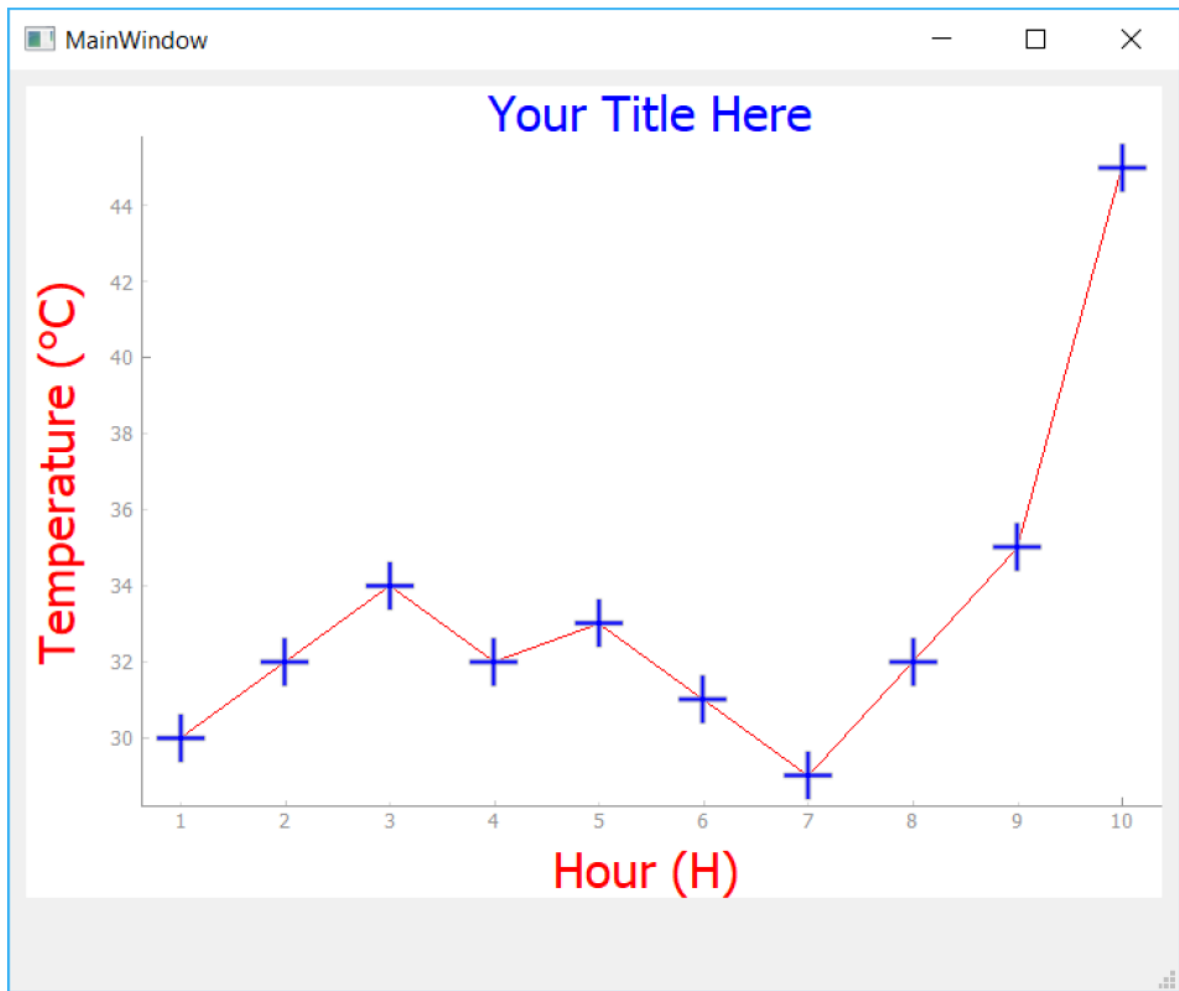


图222：自定义样式的坐标轴标签

图例

除了坐标轴和图例标题外，您通常还希望显示一个图例，用于标识特定线条所代表的内容。这在您开始向图表中添加多条线条时尤为重要。向图表添加图例可以通过调用 `PlotWidget` 的 `.addLegend` 方法实现，但在此之前，您需要在调用 `.plot()` 方法时为每条线条提供一个名称。

下面的示例将我们使用 `.plot()` 绘制的线条命名为“Sensor 1”。该名称将在图例中用于标识该线条。

```
self.graphwidget.plot(hour, temperature, name = "Sensor 1", pen = NewPen,  
symbol='+', symbolSize=30, symbolBrush=('b'))  
self.graphwidget.addLegend()
```

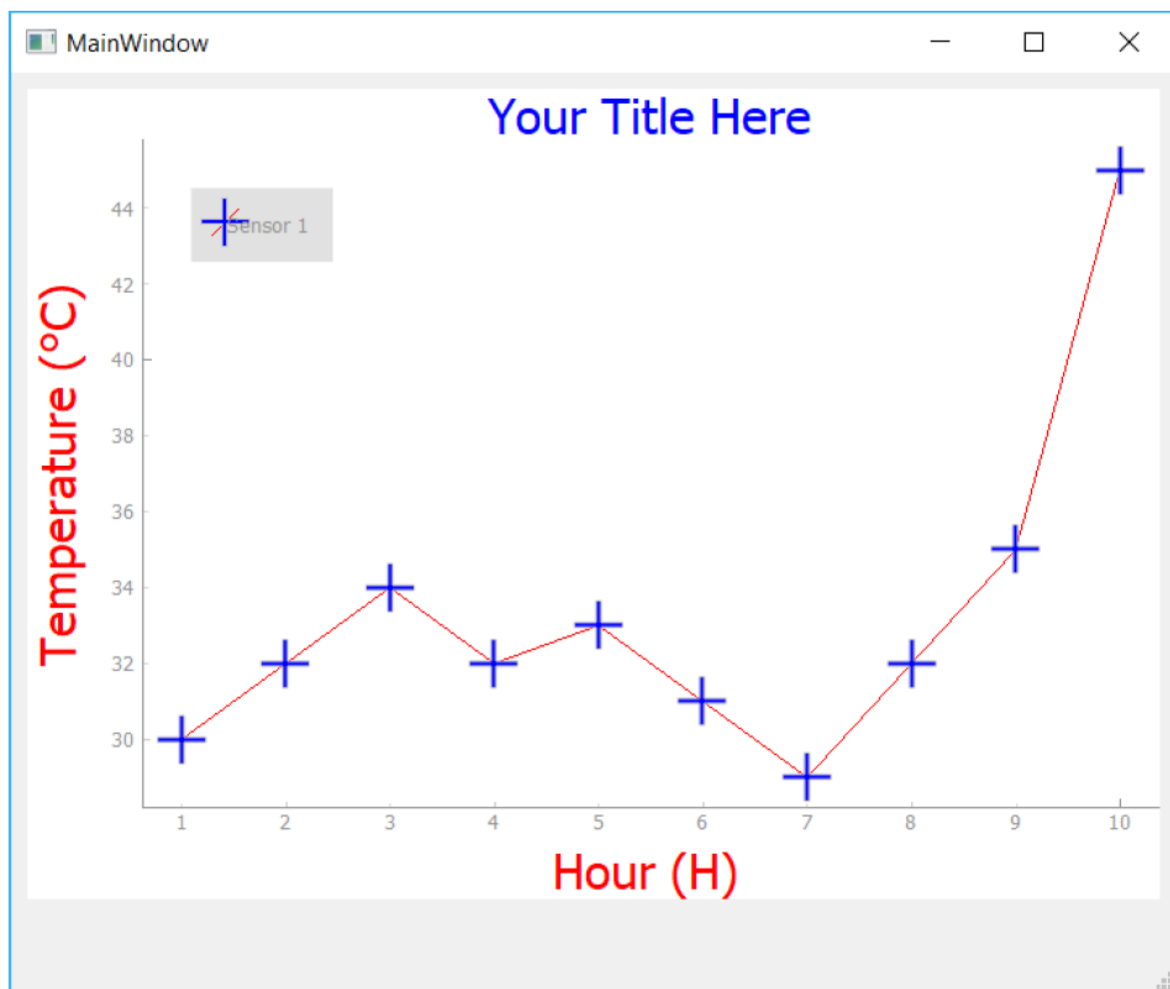


图223：带有图例的图表，显示单个项目。



图例默认显示在左上角。如果您希望移动图例，可以轻松地将图例拖动到其他位置。您还可以通过在创建图例时将一个2元组传递给 `offset` 参数来指定默认偏移量。

背景网格

添加背景网格可以使您的图表更易于阅读，尤其是在试图比较相对的 `x` 和 `y` 值时。您可以通过调用 `PlotWidget` 上的 `.showGrid` 方法来启用图表的背景网格。您可以独立地切换 `x` 和 `y` 网格。

以下内容将为X轴和Y轴创建网格。

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

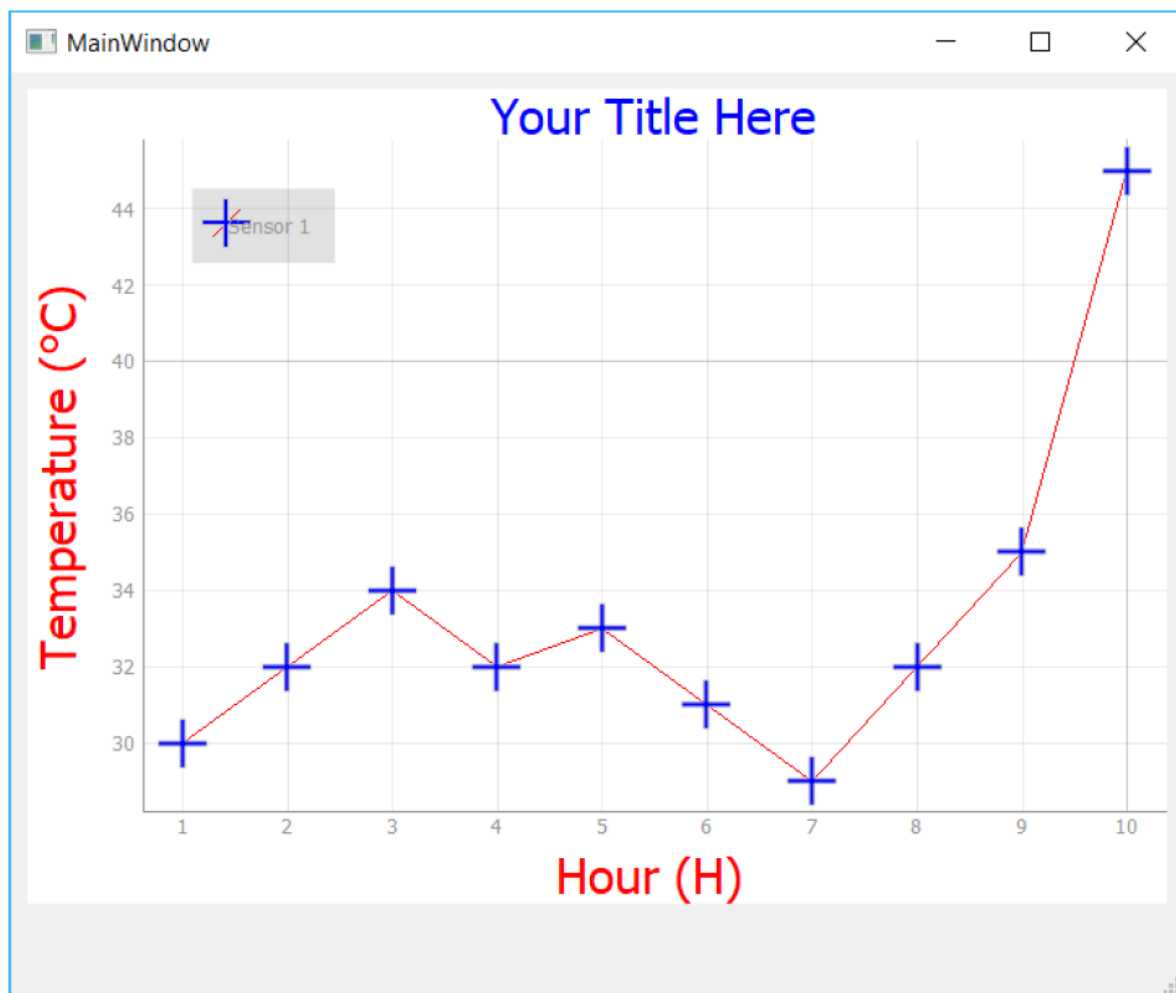


图224：背景网格

设置坐标轴的范围

有时，限制在图表上显示的数据范围，或将坐标轴锁定在固定范围（例如已知的最小值和最大值范围）可能是有用的。在 PyQtGraph 中，可以通过调用 `.setXRange()` 和 `.setYRange()` 方法实现这一点。这些方法会强制图表仅显示每个坐标轴上指定范围内的数据。

下面我们设置两个范围，每个轴一个。第一个参数是最小值，第二个参数是最大值。

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

可选的填充参数会使范围设置得比指定的范围更大，具体取决于指定的百分比（默认值在0.02到0.1之间，具体取决于视图框的大小）。如果您想完全移除这个填充，请传入0。

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

到目前为止的完整代码如下所示：

```
import sys

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph
```

```

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

        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]

        # 将背景颜色设置为白色
        self.graphwidget.setBackground("w")

        # 添加标题
        self.graphwidget.setTitle(
            "Your Title Here", color="b", size="30pt"
        )
        # 添加坐标轴标题
        styles = {"color": "#f00", "font-size": "20px"}
        self.graphwidget.setLabel("left", "Temperature (°C)", **
                                   styles)
        self.graphwidget.setLabel("bottom", "Hour (H)", **styles)
        # 添加图例
        self.graphwidget.addLegend()
        # 添加背景网格
        self.graphwidget.showGrid(x=True, y=True)
        # 设置范围
        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=30,
            symbolBrush="b",
        )

app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()

```

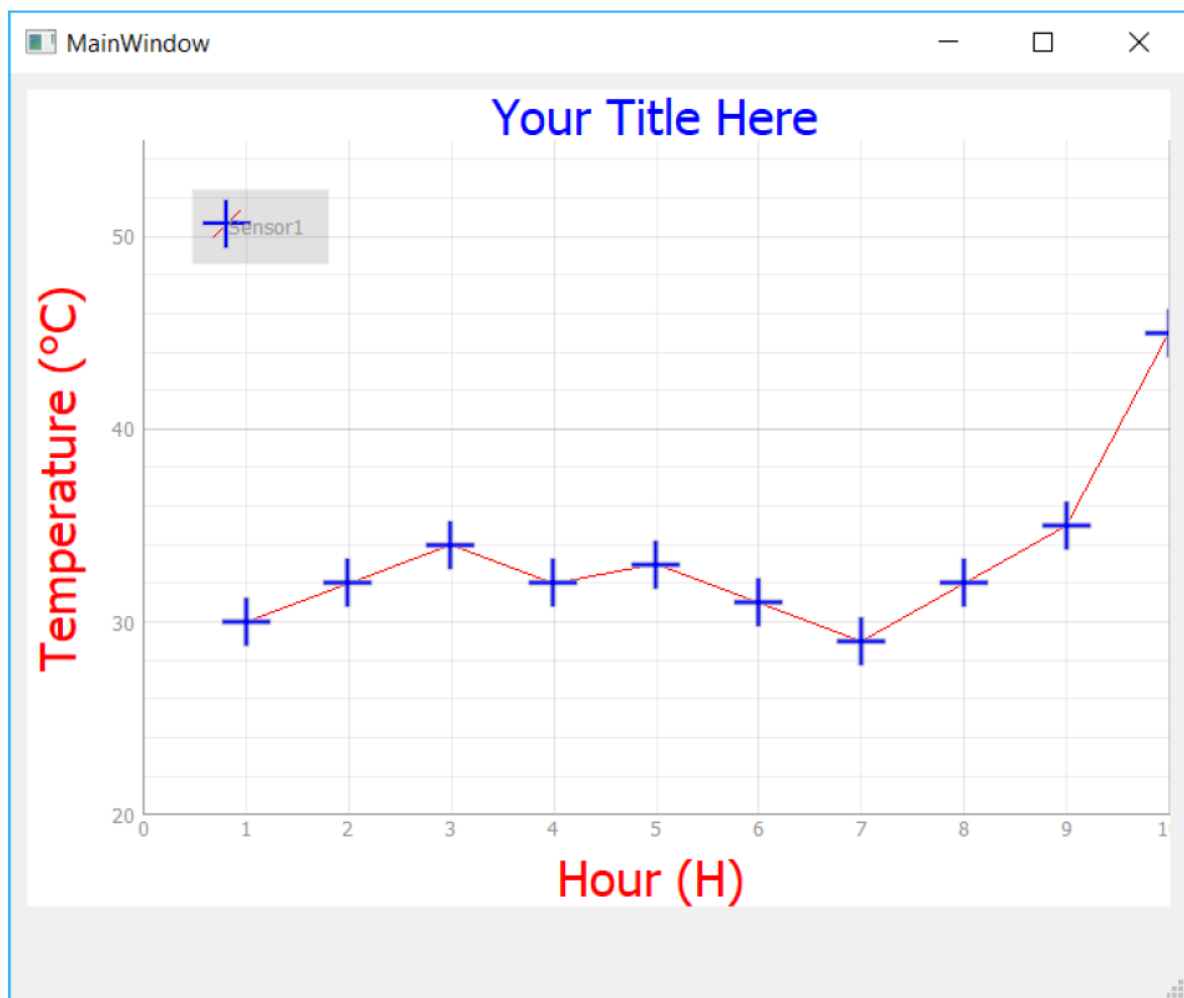


图225：限制轴的范围。

绘制多条线

绘图通常涉及多条线。在 PyQtGraph 中，这就像在同一个 `PlotWidget` 上多次调用 `.plot()` 一样简单。在以下示例中，我们将绘制两条类似的数据线，每条线使用相同的线样式、厚度等，但更改线颜色。

为了简化这一过程，我们可以在 `MainWindow` 上创建自己的自定义绘图方法。该方法接受用于绘图的 `x` 和 `y` 参数、用于图例的线条名称以及颜色参数。我们使用该颜色同时设置线条和标记的颜色。

```
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=(color))
```

要绘制单独的线条，我们将创建一个名为 `temperature_2` 的新数组，并用与 `temperature`（现在为 `temperature_1`）类似的随机数填充它。将这些线条并排绘制，我们可以将它们进行比较。现在，您可以调用 `plot` 函数两次，这将在图表上生成两条线。

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

Listing 224. `plotting/pyqtgraph_5.py`

```
import sys
```

```

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph

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

        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]

        # 将背景颜色设置为白色
        self.graphwidget.setBackground("w")

        # 添加标题
        self.graphwidget.setTitle(
            "Your Title Here", color="b", size="30pt"
        )

        # 添加坐标轴标题
        styles = {"color": "#f00", "font-size": "20px"}
        self.graphwidget.setLabel("left", "Temperature (°C)", **
                                   styles)
        self.graphwidget.setLabel("bottom", "Hour (H)", **styles)

        # 添加图例
        self.graphwidget.addLegend()

        # 添加背景网格
        self.graphwidget.showGrid(x=True, y=True)

        # 设置范围
        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=(color),
        )

app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
app.exec()

```

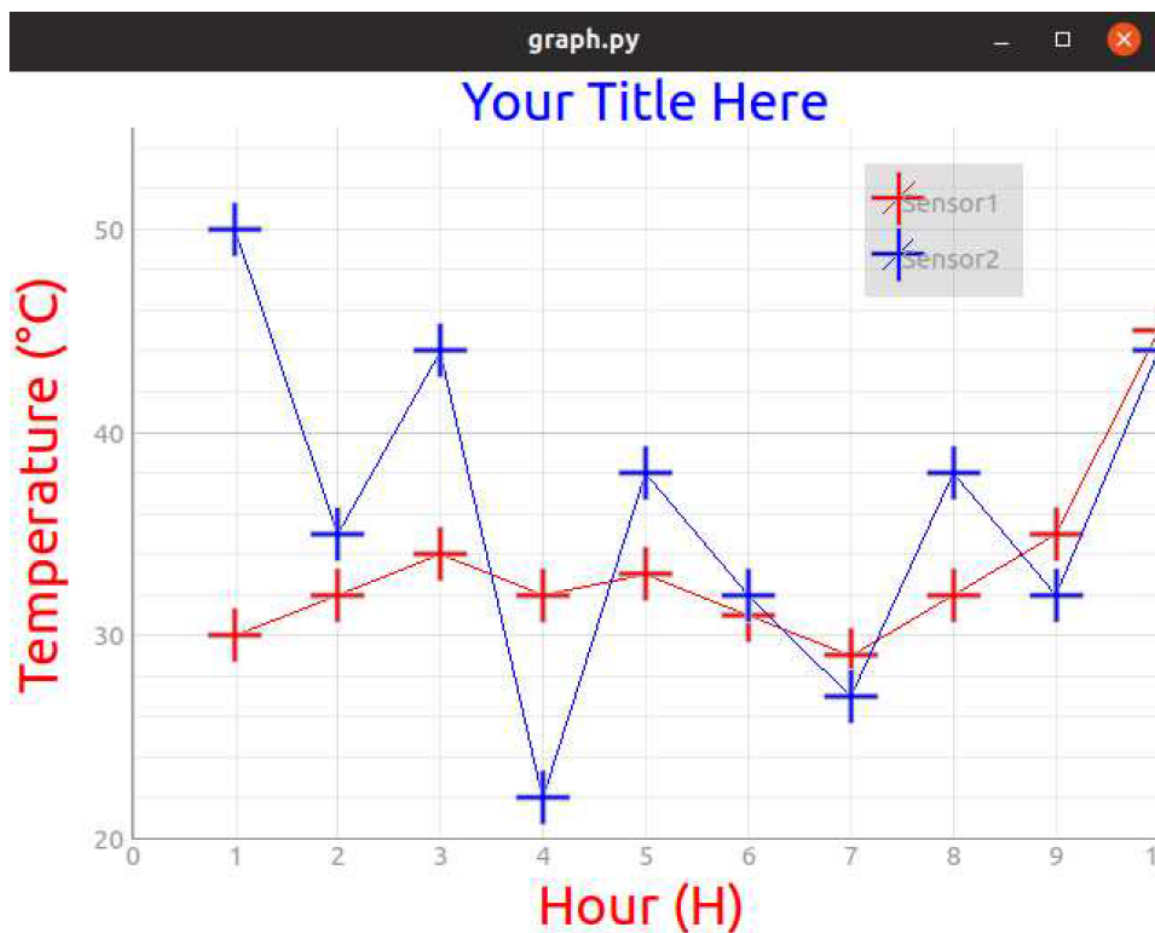



图226：包含两条线的图表



您可以尝试使用此功能，自定义您的标记、线条宽度、颜色及其他参数。

清空图表

最后，有时您可能需要定期清除并刷新绘图。您可以通过调用 `.clear()` 方法轻松实现这一点。

```
self.graphwidget.clear()
```

这将从图中删除线条，但保持所有其他属性不变。

更新图表

虽然您可以简单地清除绘图区域并重新绘制所有元素，但这意味着Qt必须销毁并重新创建所有 `QGraphicsScene` 对象。对于小型或简单的图形，这可能不会被注意到，但如果您想创建高性能的流式图形，最好直接更新数据。PyQtGraph 会获取新数据并更新绘制的线条以匹配，而不会影响图形中的其他元素。

要更新一条线，我们需要获取该线对象的引用。该引用在首次使用 `.plot` 方法创建线时返回，我们可以将其存储在变量中。请注意，这是对线对象的引用，而非对绘图的引用。

```
my_line_ref = graphwidget.plot(x, y)
```

一旦我们获得了引用，更新图表只需调用 `.setData` 方法，将新数据应用到该引用即可。

Listing 225. *plotting/pyqtgraph_6.py*

```
import sys
from random import randint

from PyQt6 import QtWidgets
import pyqtgraph as pg # 在导入Qt之后导入PyQtGraph

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

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

        self.x = list(range(100)) # 100个时间点
        self.y = [
            randint(0, 100) for _ in range(100)
        ] # 100个数据点

        self.graphwidget.setBackground("w")

        pen = pg.mkPen(color=(255, 0, 0))
        self.data_line = self.graphwidget.plot(
            self.x, self.y, pen=pen
        ) #1

        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:] # 移除第一个 y 元素.
        self.x.append(
            self.x[-1] + 1
        ) # 添加一个比上一个值大1的新值.
        self.y = self.y[1:] # 删除第一个
        self.y.append(randint(0, 100)) # 添加一个新的随机值.
        self.data_line.setData(self.x, self.y) # 更新数据.
```

1. 这里我们引用了之前绘制的线，并将它存储为 `self.data_line`。

我们使用 `QTimer` 每 50 毫秒更新一次数据，将触发器设置为调用自定义的槽方法

`update_plot_data`，在那里我们将更改数据。我们在 `__init__` 块中定义此计时器，因此它会自动启动。

如果运行该应用程序，您应该会看到一个图表，其中随机数据快速向左滚动，X 值也会同步更新并滚动，仿佛在流式传输数据。您可以用自己的真实数据替换随机数据，例如从实时传感器读数或 API 中获取。PyQtGraph 性能足够强大，可以支持使用此方法创建多个图表。

总结

在本章中，我们学习了如何使用 PyQtGraph 绘制简单的图表，并自定义线条、标记和标签。要全面了解 PyQtGraph 的方法和功能，请参阅 [PyQtGraph文档及API参考](#)。 [PyQtGraph在GitHub上的仓库](#) 中还包含了一系列更复杂的示例图表，这些示例位于Plotting.py文件中（如下所示）。

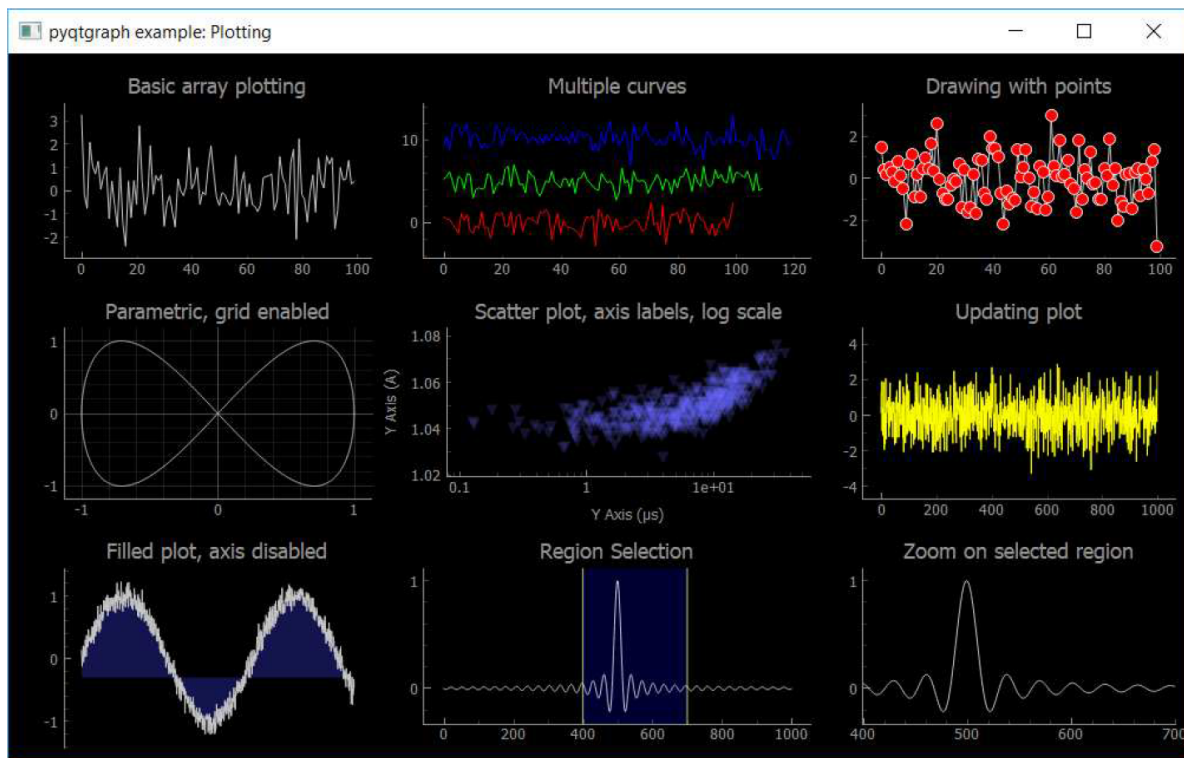


图227：示例图表摘自PyQtGraph文档。

30. 使用 Matplotlib 进行数据可视化

在前一部分中，我们介绍了如何使用 PyQtGraph 在 PyQt6 中进行绘图。该库使用 Qt 基于向量的 QGraphicsScene 来绘制图表，并提供了交互式和高性能绘图的出色接口。

然而，还有另一个用于 Python 的绘图库，它被更广泛地使用，并提供了更丰富的绘图种类——[Matplotlib](#)。如果您正在将现有的数据分析工具迁移到 PyQt6 图形用户界面，或者您只是想使用 Matplotlib 提供的各种绘图功能，那么您需要了解如何将 Matplotlib 绘图纳入您的应用程序。

在本章中，我们将介绍如何在 PyQt6 应用程序中嵌入 Matplotlib 图表。



许多其他 Python 库（如 [seaborn](#) 和 [pandas](#)）也使用 Matplotlib 进行绘图。这些绘图可以像这里示例中一样嵌入到 PyQt6 中，并且在绘图时传递的轴引用。本章末尾有一个 pandas 的示例。

安装 Matplotlib

以下示例假设您已安装 Matplotlib。如果未安装，您可以使用 `pip` 进行安装。

撰写本文时，PyQt6 还非常新。有一个实验性分支，其中包含[Qt6 支持](#)，您可以使用它 —

```
pip install git+https://github.com/anntzer/matplotlib.git@qt6
```

一个简单的例子

以下最简单的示例设置了一个 Matplotlib 画布 `FigureCanvasQTAgg`，该画布创建了 `Figure` 并向其中添加了一组轴。该画布对象也是一个 `QWidget`，因此可以像其他 Qt 控件一样直接嵌入到应用程序中。

Listing 226. plotting/matplotlib_1.py

```
import sys

from PyQt6 import QtWidgets # 在导入matplotlib之前先导入PyQt6。

import matplotlib
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.figure import Figure

matplotlib.use("QtAgg")

class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        super().__init__(fig)

class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()
        # 创建 matplotlib FigureCanvasQTAgg 对象，该对象定义了一组坐标轴，即 self.axes.
        sc = MplCanvas(self, width=5, height=4, dpi=100)
        sc.axes.plot([0, 1, 2, 3, 4], [10, 1, 20, 3, 40])
        self.setCentralWidget(sc)

        self.show()

app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()
```

在这种情况下，我们使用 `.setCentralWidget()` 将 `MplCanvas` 控件添加到窗口作为中央控件。这意味着它将占据整个窗口，并随窗口一起调整大小。绘制的数据 `[0, 1, 2, 3, 4]`, `[10, 1, 20, 3, 40]` 以两个数字列表（分别表示 x 和 y 坐标）的形式提供，符合 `.plot` 方法的要求。

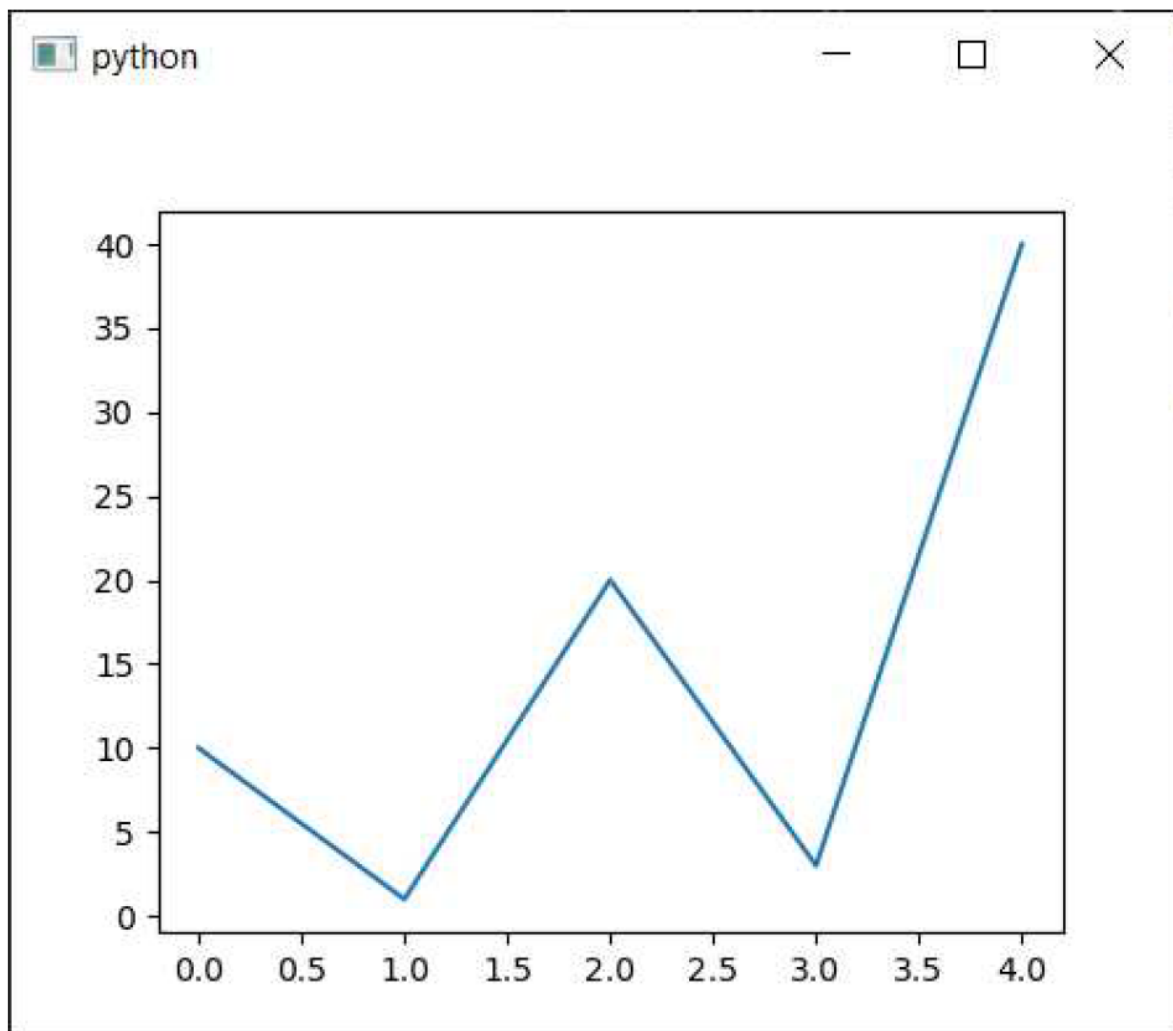


图228：一个简单的图表

图表控制

在 PyQt6 中显示的 Matplotlib 图实际上是由 Agg 后端渲染为简单的（位图）图像。

`FigureCanvasQTAgg` 类封装了这个后端，并将生成的图像显示在 Qt 控件上。这种架构的效果是，Qt 不知道线和其他绘图元素的位置，只知道在控件上点击和鼠标移动的 x、y 坐标。

然而，Matplotlib 内置了对 Qt 鼠标事件的处理和将其转换为图上的交互的支持。这可以通过一个自定义工具栏来控制，该工具栏可以与图一起添加到您的应用程序中。在本节中，我们将介绍如何添加这些控件，以便我们能够缩放、平移和从嵌入的 Matplotlib 图中获取数据。

完整的代码如下所示，它导入了工具栏控件 `NavigationToolbar2QT`，并将其添加到 `QVBoxLayout` 中的界面中。

Listing 227. `plotting/matplotlib_2.py`

```
import sys
from PyQt6 import QtWidgets # 在导入matplotlib之前先导入PyQt6
import matplotlib
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.backends.backend_qtagg import (
    NavigationToolbar2QT as NavigationToolbar,
)
from matplotlib.figure import Figure

matplotlib.use("QtAgg")
```

```

class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        super().__init__(fig)

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

        sc = MplCanvas(self, width=5, height=4, dpi=100)
        sc.axes.plot([0, 1, 2, 3, 4], [10, 1, 20, 3, 40])

        # 创建工具栏，将画布作为第一个参数传递，父窗口(self, the MainWindow)作为第二个参
        # 数。
        toolbar = NavigationToolbar(sc, self)

        layout = QtWidgets.QVBoxLayout()
        layout.addWidget(toolbar)
        layout.addWidget(sc)

        # 创建一个占位符控件来容纳我们的工具栏和画布。
        widget = QtWidgets.QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)

        self.show()

app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()

```

我们将逐一说明这些变更。

首先，我们从 `matplotlib.backends.backend_qt5agg.NavigationToolbar2QT` 导入工具栏控件，并将其重命名为更简单的名称 `NavigationToolbar`。我们通过调用 `NavigationToolbar` 并传入两个参数来创建工具栏实例，第一个参数是画布对象 `sc`，第二个参数是工具栏的父对象，在本例中是 `MainWindow` 对象 `self`。传入画布对象可将创建的工具栏与之关联，从而实现对其的控制。生成的工具栏对象被存储在变量 `toolbar` 中。

我们需要在窗口中添加两个控件，一个在另一个上方，因此我们使用一个 `QVBoxLayout`。首先，我们将工具栏控件 `toolbar` 和画布控件 `sc` 添加到此布局中。最后，我们将此布局设置到我们的简单控件布局容器中，该容器被设置为窗口的中央控件。

运行上述代码将生成以下窗口布局，显示在底部的绘图和顶部的工具栏作为控件。

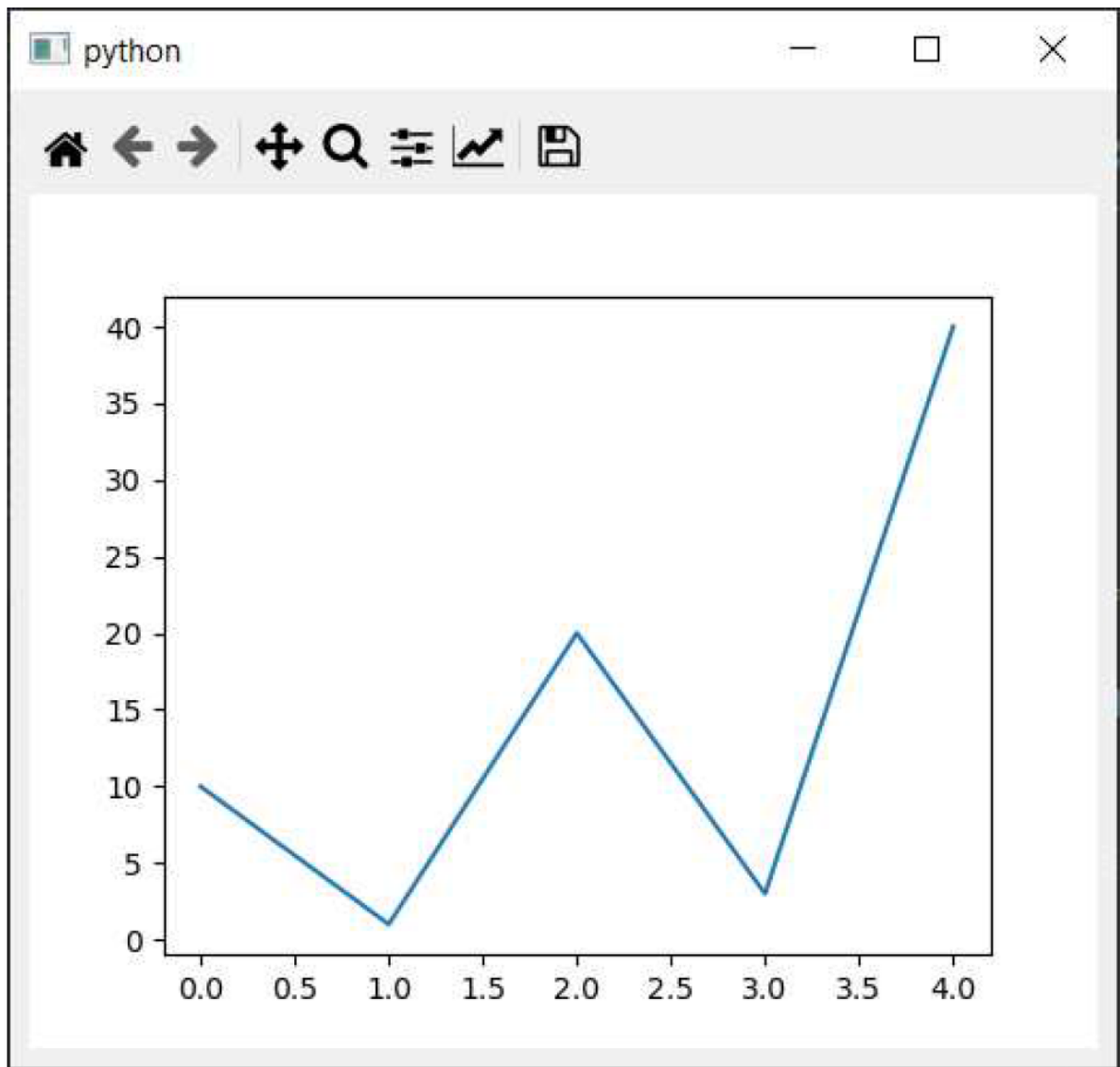


图229：带工具栏的Matplotlib画布

NavigationToolbar2QT 提供的按钮可用于控制以下操作：

- 首页、后退/前进、平移与缩放，用于在图中导航。后退/前进按钮可用于在导航步骤之间前后移动，例如先放大再点击后退按钮将返回上一个缩放级别。首页按钮可返回图的初始状态。
- 绘图边距/位置配置，可调整绘图在窗口内的位置。
- 轴/曲线样式编辑器，您可以在其中修改图标题和轴刻度，以及设置图线颜色和线样式。颜色选择使用平台默认的颜色选择器，允许选择任何可用的颜色。
- 保存，将生成的图形保存为图像（支持所有Matplotlib支持的格式）。

以下列出了其中一些配置设置：

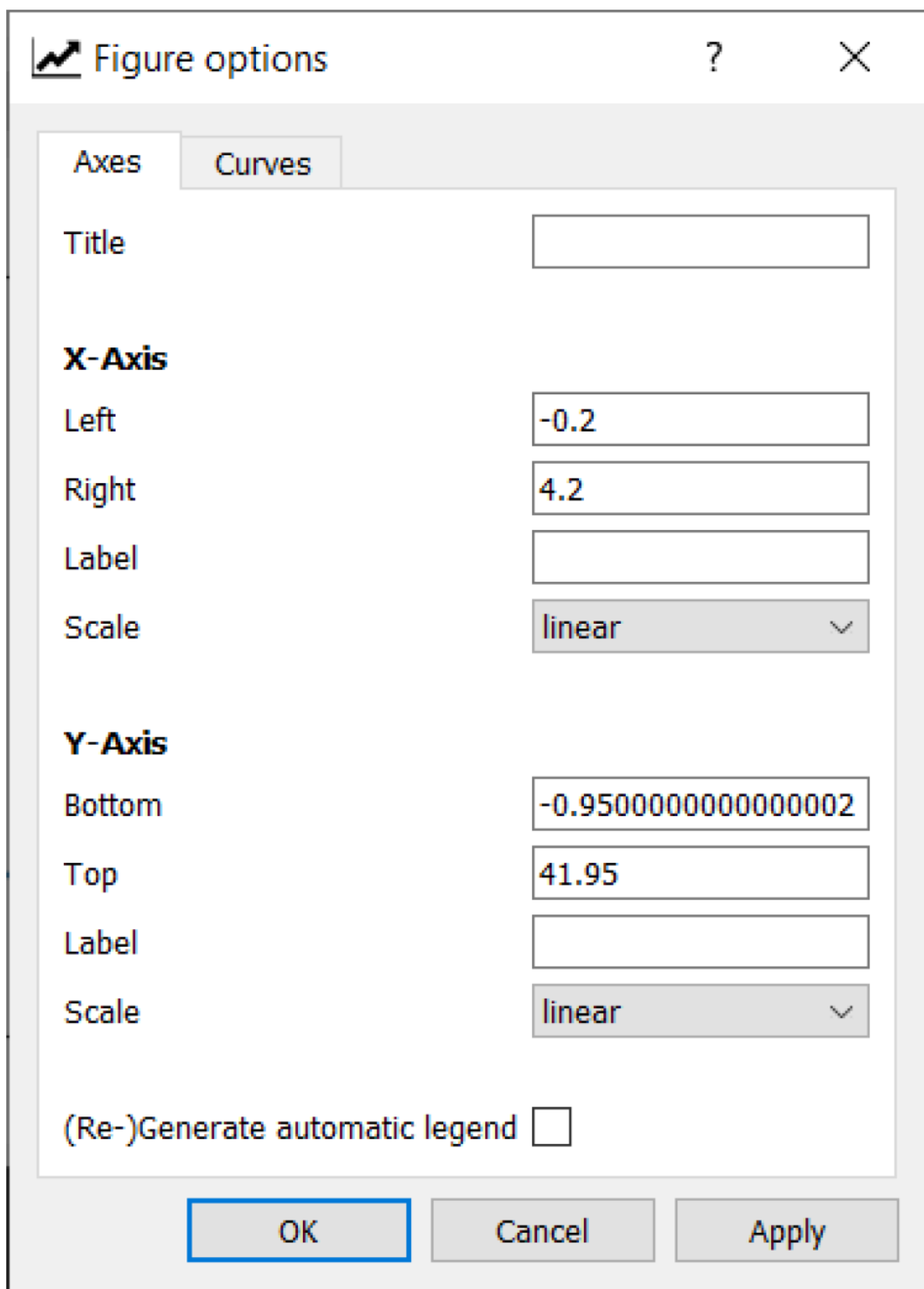
The image shows a 'Figure options' dialog box with a title bar containing a line graph icon, the text 'Figure options', and standard window controls (help, close). The dialog has two tabs: 'Axes' (selected) and 'Curves'. Under the 'Axes' tab, there are settings for the title, X-axis, and Y-axis. The X-axis settings include 'Left' (value: -0.2), 'Right' (value: 4.2), 'Label' (empty), and 'Scale' (dropdown: linear). The Y-axis settings include 'Bottom' (value: -0.95000000000000002), 'Top' (value: 41.95), 'Label' (empty), and 'Scale' (dropdown: linear). At the bottom, there is a checkbox for '(Re-)Generate automatic legend' which is currently unchecked. The dialog concludes with 'OK', 'Cancel', and 'Apply' buttons.

Figure options

?

×

AxesCurves

Title

X-Axis

Left

-0.2

Right

4.2

Label

Scale

linear

Y-Axis

Bottom

-0.95000000000000002

Top

41.95

Label

Scale

linear

(Re-)Generate automatic legend ☐

OK

Cancel

Apply

图230: Matplotlib 图表选项

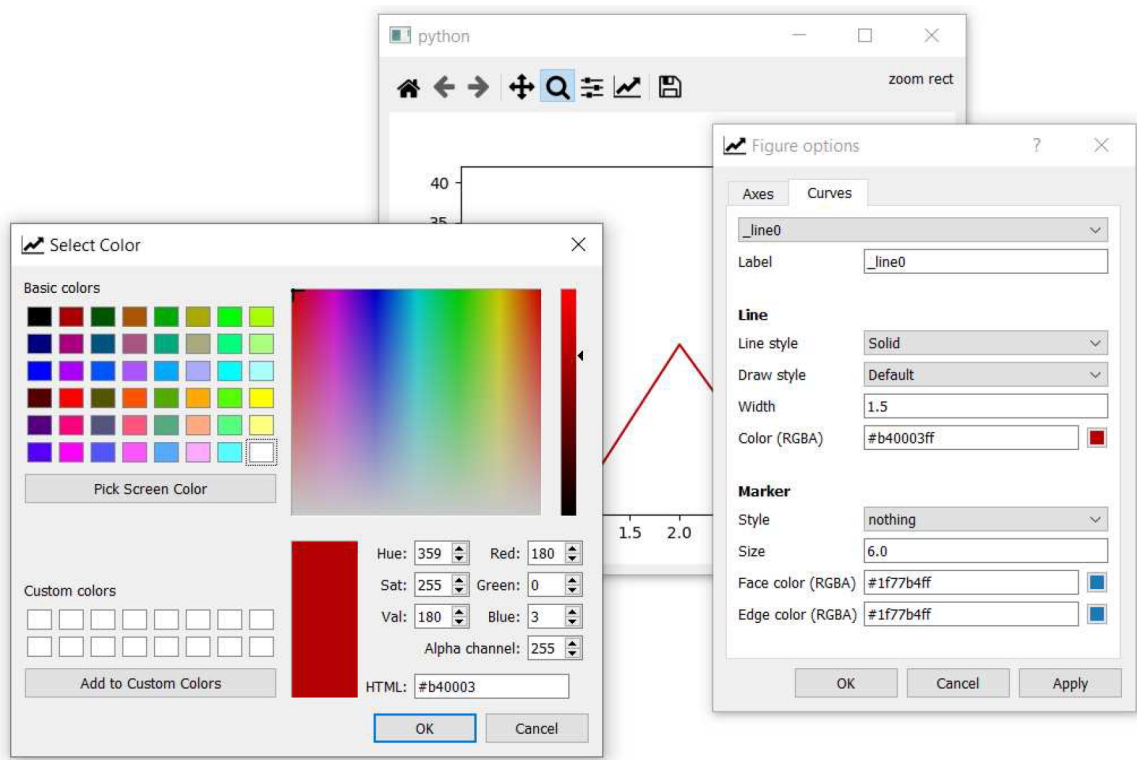


图231: Matplotlib 曲线选项

有关如何操作和配置Matplotlib图表的更多信息，请参阅官方 [Matplotlib工具栏文档](#)。

更新图表

在应用程序中，您经常需要更新图表中显示的数据，无论是响应用户的输入还是来自 API 的更新数据。在 Matplotlib 中，有两种方法可以更新图表，您可以选择：

1. 清除并重新绘制画布（更简单，但速度较慢）
2. 通过保留绘制线的引用并更新数据

如果性能对您的应用程序很重要，建议您选择后者，但前者更简单。我们先从简单的清除并重绘方法开始：

清除并重新绘制

Listing 228. plotting/matplotlib_3.py

```
import sys
from PyQt6 import QtWidgets # 在导入matplotlib之前先导入PyQt6
import matplotlib
from matplotlib.backends.backend_qtagg import FigureCanvasQTAgg
from matplotlib.backends.backend_qtagg import (
    NavigationToolbar2QT as NavigationToolbar,
)
from matplotlib.figure import Figure

matplotlib.use("QtAgg")

class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
```

```

super().__init__(fig)

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

        self.canvas = MplCanvas(self, width=5, height=4, dpi=100)
        self.setCentralWidget(self.canvas)

        n_data = 50
        self.xdata = list(range(n_data))
        self.ydata = [random.randint(0, 10) for i in range(n_data)]
        self.update_plot()

        self.show()

        # 设置一个定时器，通过调用 update_plot 函数触发重新绘制。
        self.timer = QtCore.QTimer()
        self.timer.setInterval(100)
        self.timer.timeout.connect(self.update_plot)
        self.timer.start()

    def update_plot(self):
        # 删除第一个 y 元素，并添加一个新的 y 元素。
        self.ydata = self.ydata[1:] + [random.randint(0, 10)]
        self.canvas.axes.cla() # 清空画布。
        self.canvas.axes.plot(self.xdata, self.ydata, "r")
        # 触发画布更新并重新绘制。
        self.canvas.draw()

app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()

```

在此示例中，我们将绘图操作移至 `update_plot` 方法中，以保持其独立性。在此方法中，我们对 `ydata` 数组的第一个值进行 `[1:]` 操作移除，然后追加一个0到10之间的随机整数。此操作会将数据向左滚动。

要重新绘制，我们只需调用 `axes.cla()` 清除坐标轴（整个画布），然后调用 `axes.plot(...)` 重新绘制数据，包括更新后的值。然后，通过调用 `canvas.draw()` 将生成的画布重新绘制到控件上。

`update_plot` 方法每 100 毫秒通过 `qtimer` 调用一次。`clear-and-refresh` 方法足够快，可以以这种速度保持绘图更新，但如我们将看到的，随着速度的提高，它会出现问题。

就地重绘

更新绘制线条所需的更改相对较少，仅需添加一个变量来存储并获取绘制线条的引用。更新后的 `MainWindow` 代码如下所示：

Listing 229. plotting/matplotlib_4.py

```

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

```

```

super().__init__()
self.canvas = MplCanvas(self, width=5, height=4, dpi=100)
self.setCentralWidget(self.canvas)
n_data = 50
self.xdata = list(range(n_data))
self.ydata = [random.randint(0, 10) for i in range(n_data)]

# 我们需要将绘制的线条的引用保存在某个地方，以便我们可以将新数据应用到它上。
self._plot_ref = None
self.update_plot()

self.show()

# 设置一个定时器，通过调用 update_plot 函数触发重新绘制。
self.timer = QtCore.QTimer()
self.timer.setInterval(100)
self.timer.timeout.connect(self.update_plot)
self.timer.start()

def update_plot(self):
    # 删除第一个 y 元素，并添加一个新的元素。
    self.ydata = self.ydata[1:] + [random.randint(0, 10)]

    # 注意：我们不再需要清除轴。。
    if self._plot_ref is None:
        # 第一次没有图表参考，所以做一个正常的图表。
        # .plot 返回一个线性引用列表，由于我们只获取一个，可以直接取第一个元素。
        plot_refs = self.canvas.axes.plot(
            self.xdata, self.ydata, "r"
        )
        self._plot_ref = plot_refs[0]
    else:
        # 我们有一个引用，可以使用它来更新该行的数据。
        self._plot_ref.set_ydata(self.ydata)
    # 触发画布更新并重新绘制。
    self.canvas.draw()

```

首先，我们需要一个变量来保存要更新的绘制线条的引用，这里我们将其命名为 `_plot_ref`。我们初始化 `self._plot_ref` 为 `None`，这样我们可以在后续代码中检查其值以确定线条是否已绘制——如果值仍然为 `None`，则表示线条尚未绘制。



如果您需要绘制多条线，您可能希望使用列表或字典数据结构来存储多个引用，并跟踪每条线的具体信息。

最后，我们像之前一样更新 `ydata` 数据，将其向左旋转并添加一个新的随机值。然后我们可以选择：

1. 如果 `self._plot_ref` 为 `None`（即尚未绘制该线），则绘制该线，并将引用存储在 `self._plot_ref` 中。

2. 通过调用 `self._plot_ref.set_ydata(self.ydata)` 来更新当前的线条。

在调用 `.plot` 时，我们会获得绘制线的引用。然而，`.plot` 方法会返回一个列表（以支持单次 `.plot` 调用绘制多条线的情况）。在我们的情况下，我们只绘制一条线，因此我们只需获取该列表中的第一个元素——一个 `Line2D` 对象。为了将这个单一值赋给我们的变量，我们可以先将它赋给一个临时变量 `plot_refs`，然后将第一个元素赋给我们的 `self._plot_ref` 变量。

```
plot_refs = self.canvas.axes.plot(self.xdata, self.ydata, 'r')
self._plot_ref = plot_refs[0]
```

您还可以使用元组解包，从列表中提取第一个（也是唯一）元素：

```
self._plot_ref, = self.canvas.axes.plot(self.xdata, self.ydata, 'r')
```

如果您运行生成的代码，在当前速度下，这种方法与之前的方法在性能上不会有明显差异。然而，如果您尝试以更快的速度更新图表（例如每10毫秒一次），您会发现清除图表并重新绘制所需的时间更长，且更新速度无法跟上计时器。这种性能差异是否足以在您的应用程序中产生影响，取决于您正在构建的内容，并且您应该权衡保持和管理绘制线条的引用所带来的额外复杂性。

从 Pandas 中嵌入图表

Pandas 是一个专注于处理表格（数据框）和系列数据结构的 Python 包，对于数据分析工作流程特别有用。它内置了对 Matplotlib 绘图的支持，本文将简要介绍如何将这些绘图嵌入到 PyQt6 中。通过此方法，您可以开始构建基于 Pandas 的 PyQt6 数据分析应用程序

Pandas 的绘图函数可直接从 `DataFrame` 对象访问。该函数的签名较为复杂，提供了大量选项来控制绘图的具体方式。

```
DataFrame.plot(
    x=None, y=None, kind='line', ax=None, subplots=False,
    sharex=None, sharey=False, layout=None, figsize=None,
    use_index=True, title=None, grid=None, legend=True, style=None,
    logx=False, logy=False, loglog=False, xticks=None, yticks=None,
    xlim=None, ylim=None, rot=None, fontsize=None, colormap=None,
    table=False, yerr=None, xerr=None, secondary_y=False,
    sort_columns=False, **kwargs
)
```

我们最感兴趣的参数是 `ax`，它允许我们传入自己的 `matplotlib.Axes` 实例，Pandas 将在此实例上绘制 `DataFrame`。

Listing 230. plotting/matplotlib_5.py

```
import sys
from PyQt6 import (
    QtCore,
    QtWidgets,
) # 在导入matplotlib之前先导入PyQt6
import matplotlib
import pandas as pd
from matplotlib.backends.backend_qtagg import FigureCanvasQTagg
from matplotlib.figure import Figure
```

```

matplotlib.use("QtAgg")

class MplCanvas(FigureCanvasQTAgg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
        super().__init__(fig)

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

        # 创建 matplotlib FigureCanvasQTAgg 对象, 该对象定义了一组坐标轴, 即 self.axes.
        sc = MplCanvas(self, width=5, height=4, dpi=100)
        # 创建一个包含简单数据和列名的 pandas DataFrame.
        df = pd.DataFrame(
            [
                [0, 10],
                [5, 15],
                [2, 20],
                [15, 25],
                [4, 10],
            ],
            columns=["A", "B"],
        )
        # 绘制 pandas DataFrame, 传入 matplotlib Canvas 坐标轴.
        df.plot(ax=sc.axes)

        self.setCentralWidget(sc)
        self.show()

app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()

```

这里的关键步骤是在调用 `DataFrame` 的 `plot` 方法时传入画布坐标轴, 即在行 `df.plot(ax=sc.axes)` 中传入。您可以使用相同的模式来随时更新图表, 不过需要注意的是, Pandas会清空并重新绘制整个画布, 这意味着它并不适合高性能绘图。

通过Pandas生成的结果图如下所示:

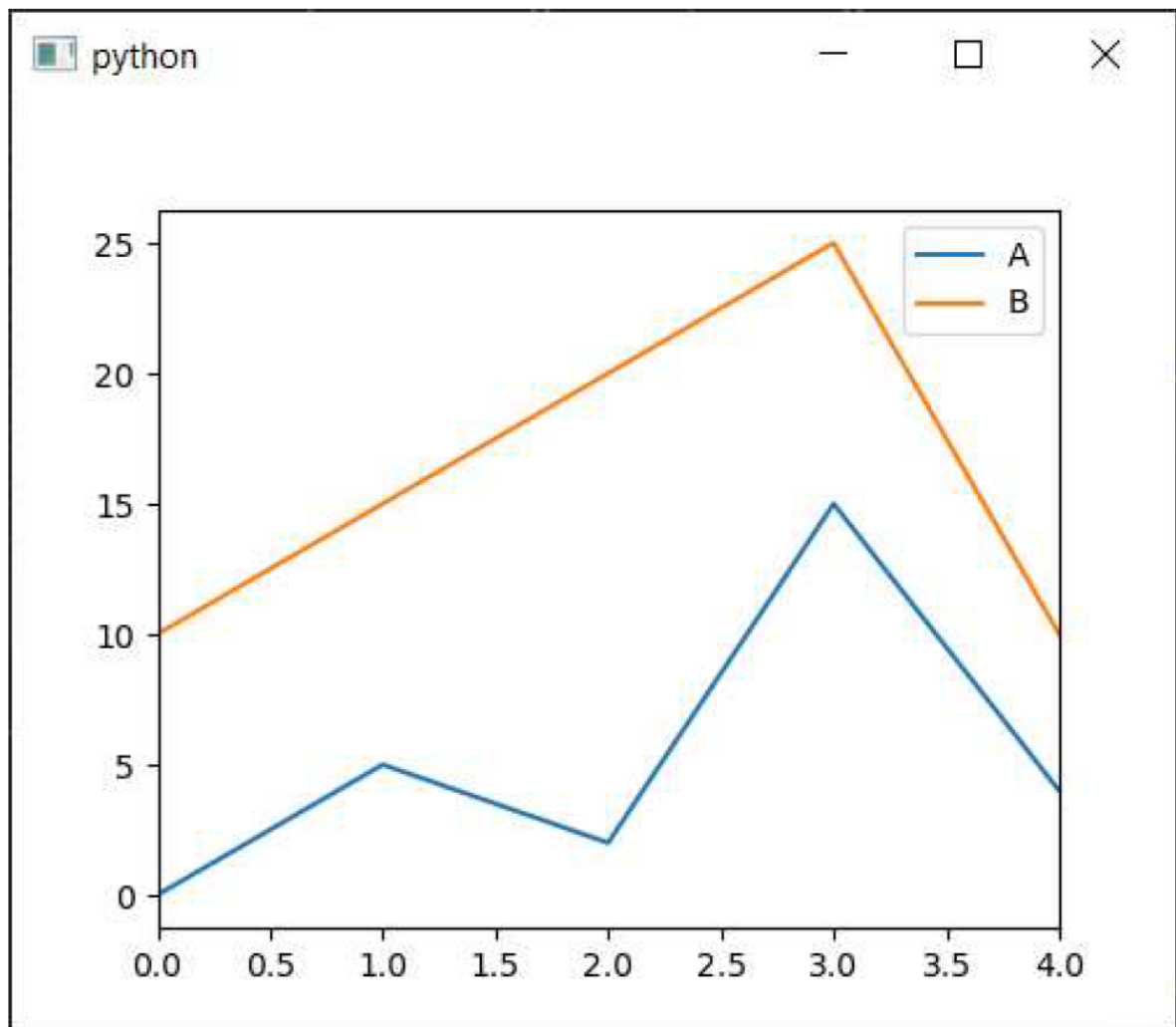


图232: 使用matplotlib Canvas生成的 pandas 图

与之前一样，您可以为使用Pandas生成的图表添加Matplotlib工具栏和控制支持，允许您实时缩放/平移并修改图表。以下代码将我们之前的工具栏示例与Pandas示例相结合：

Listing 231. *plotting/matplotlib_6.py*

```
import sys
from PyQt6 import (
    QtCore,
    QtWidgets,
) # 在导入matplotlib之前先导入PyQt6
import matplotlib
import pandas as pd
from matplotlib.backends.backend_qtagg import FigureCanvasQTagg
from matplotlib.backends.backend_qtagg import (
    NavigationToolbar2QT as NavigationToolbar,
)
from matplotlib.figure import Figure

matplotlib.use("QtAgg")

class MplCanvas(FigureCanvasQTagg):
    def __init__(self, parent=None, width=5, height=4, dpi=100):
        fig = Figure(figsize=(width, height), dpi=dpi)
        self.axes = fig.add_subplot(111)
```

```

super().__init__(fig)

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

        # 创建 matplotlib FigureCanvasQTAgg 对象, 该对象定义了一组坐标轴, 即 self.axes.
        sc = MplCanvas(self, width=5, height=4, dpi=100)
        # 创建一个包含简单数据和列名的 pandas DataFrame.
        df = pd.DataFrame(
            [
                [0, 10],
                [5, 15],
                [2, 20],
                [15, 25],
                [4, 10],
            ],
            columns=["A", "B"],
        )
        # 绘制 pandas DataFrame, 传入 matplotlib Canvas 坐标轴.
        df.plot(ax=sc.axes)

        # 创建工具栏, 将画布作为第一个参数传递, 父窗口(self, the MainWindow)作为第二个参数.
        toolbar = NavigationToolbar(sc, self)

        layout = QtWidgets.QVBoxLayout()
        layout.addWidget(toolbar)
        layout.addWidget(sc)

        # 创建一个占位符控件来容纳我们的工具栏和画布.
        widget = QtWidgets.QWidget()
        widget.setLayout(layout)

        self.setCentralWidget(sc)
        self.show()

app = QtWidgets.QApplication(sys.argv)
w = MainWindow()
app.exec()

```

运行此代码后, 您应看到以下窗口, 其中显示了一个Pandas图表嵌入在PyQt6中, 并配有Matplotlib工具栏

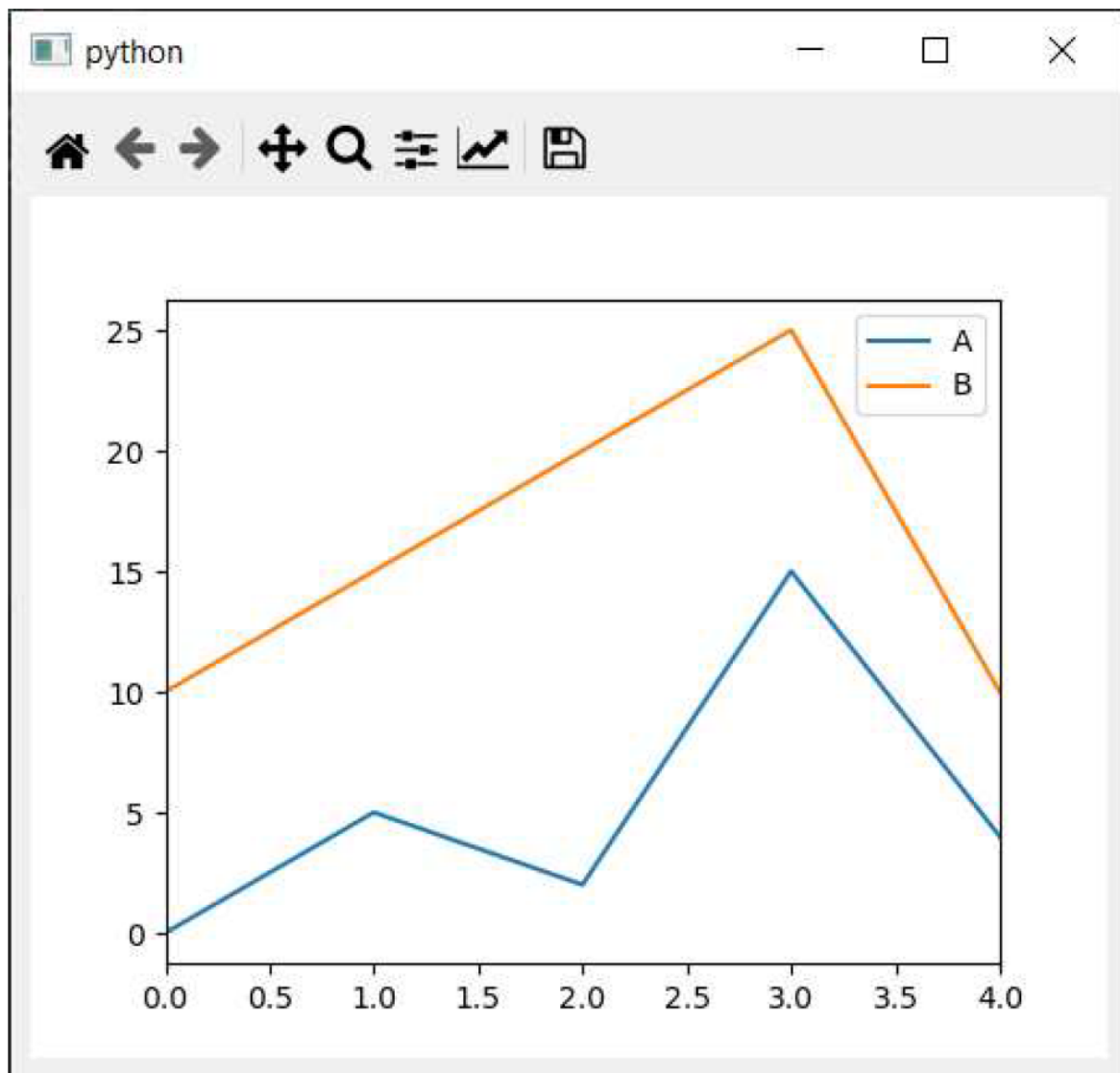


图233：使用matplotlib工具栏绘制 Pandas 图

接下来是什么？

在本章中，我们探讨了如何在 PyQt6 应用程序中嵌入 Matplotlib 图表。能够在应用程序中使用 Matplotlib 图表，使您能够从 Python 创建自定义数据分析和可视化工具。

Matplotlib 是一个庞大的库，内容过于丰富，无法在此详细展开。如果您对 Matplotlib 的绘图功能不熟悉，但想尝试使用，建议您查阅 [相关文档](#) 和 [示例图表](#)，以了解其功能范围。