Custom Widgets

Qt gives you all the standard user-interface elements you've grown up with: pushbuttons, checkboxes, drop-down lists, scrollbars, text input fields, sliders, etc.. they're all there, ready to use. And with stylesheets you can make these common UI objects look however you like.

This worksheet is about how you can go beyond the normative interface elements. Here you'll learn how to design custom widgets with unique visual appearance, behavior and user interactions.

Since all widgets extend `QWidget`, we'll need to understand some of the methods inside `QWidget` that can be overridden in your subclasses to add custom behaviors and draw unique form factors. A read-through of the documentation for `QWidget` is useful. You can find it here: https://doc.qt.io/qt-5/qwidget.html#details

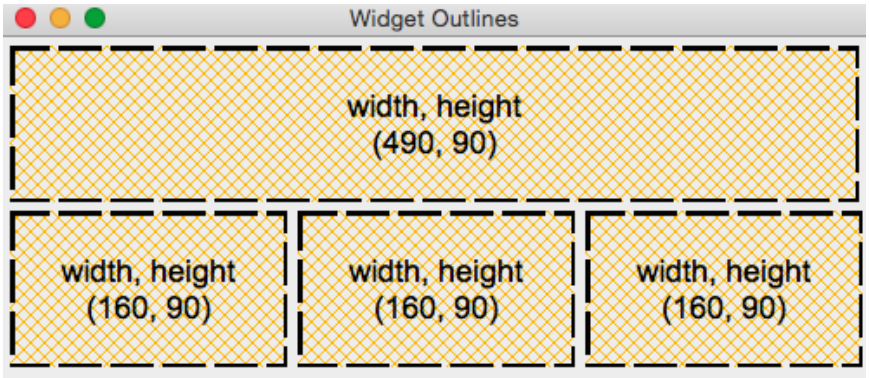
Particularly important to this worksheet are the sections on "Custom Widgets and Painting" and "Events". Qt is an "event driven" framework, and QWidget has a number of methods that respond to important events that a user interface element might encounter. Events such as *"hey Widget, redraw yourself!"* (`paintEvent()`) or *"hey Widget, the user just clicked on you"* (`mousePressEvent()`). By overriding these "event responder" methods in your subclasses, you can design a completely new species of widget.

The `paintEvent()` Method

`paintEvent()` is the method of `QWidget` that gets called every time a widget needs to draw (or redraw) itself to the screen. Nearly all child classes of `QWidget` override this method, providing the basic drawing commands for that specific widget. For example, a `QPushButton` overrides paintEvent() with instructions on drawing the basic button formula: it's rectangular shape and borders, its inner text, using alternative colors if it's pressed. The `paintEvent()` method in `QPushButton` also checks for stylesheets and, if needed, uses them to adjust its drawing instructions accordingly.

When making drawing instructions for your dream widgets, it's important to first understand that every widget is basically a rectangular canvas. The boundaries of a widget are defined by a `PyQt5.QtCore.QRect` that specifies an *(x,y) location* for the upper left corner of the widget, and a *width and height* for the widget's rectangle.* You can retrieve the bounding rectangle of a widget using the `rect()` method of `QWidget`.

** You're redefining this QRect every time you use QWidget's* `resize()` *or* `setGeometry()` *method*



Rectangles and text drawn directly to the canvas inside paintEvent()

To draw to the canvas of your widget inside `paintEvent()`,the first thing you need to do is create a `QPainter` object. `QPainter` is a special object that contains all the functionality needed to draw to a widget's canvas: e.g. pens, brushes, functions to draw rectangles and curves. `QPainter` is very similar to the toolbox you would have in a program like MSPaint or Photoshop. The constructor for `QPainter` takes one argument, the widget whose canvas you want to draw to. Usually you're creating a `QPainter` instance inside of a widget's `paintEvent()` method. The basic method skeleton will look something like this:

```
def paintEvent(self, event):
        qp = QPainter(self) # make a new QPainter that will draw to the canvas of this widget
        # Your drawing commands go here
        # For example, draw a 30x30pixel blue rectangle with its top-left corner at (10,10):
        # qp.setBrush(QBrush(Qt.blue))
        # qp.drawRect(10, 10, 30, 30)
```

Note: QPainter has a huge number of methods for drawing lines, shapes, text and images. Consult the documentation for `QPainter` *for a complete list.*

Before you draw any lines or shapes with `QPainter` you'll need to tell it what kind of line and fill styles to use. There are a number of classes in Qt used to describe the *way* to draw something. Some of the main style classes and their constants are outlined below, for a full list see the documentation.

`QPen`	Describes line style, width, color and endcaps		PyQt5.QtGui.QPen
`Qt.PenStyle`		Line style constants	PyQt5.QtCore.Qt.PenStyle
	`Qt.SolidLine`	A plain line	
	`Qt.DashLine`	A dashed line	
	`Qt.NoPen`	No line (for drawing shapes with no outline)	
`QBrush`	Describes fill color, pattern or gradient		PyQt5.QtGui.QBrush
`Qt.BrushStyle`		Brush style constants	PyQt5.QtCore.Qt.BrushStyle
	`Qt.SolidPattern`	A solid color fill	
	`Qt.NoBrush`	No fill (for drawing shape outlines)	
	`Qt.CrossPattern`	A criss-cross fill pattern	
`QFont`	Describes font family, size and style		PyQt5.QtGui.QFont
`QColor`	Describes a color		PyQt5.QtGui.QColor
`Qt.GlobalColor`		A number of predefined color constants provided by Qt	PyQt5.QtCore.Qt.GlobalColor

Let's look at an example. The following program creates the window shown in the image above. It defines a custom widget called `Outline` that draws a dotted line along its border, fills its canvas with an orange cross pattern, and draws some text in its center indicating its width and height. Then, inside a custom window (`MyWin`) four `Outline` widgets are created and positioned absolutely.

`Widget-Outlines.py`

```
1 from PyQt5.QtWidgets import *
2 from PyQt5.QtCore import Qt
3 from PyQt5.QtGui import QBrush, QPen, QColor, QFont, QPainter
4 import sys
5
6 class Outline(QWidget):
```

```
 7      def __init__(self, parent):
 8          super(Outline, self).__init__(parent)
 9          self.brush = QBrush(QColor(255, 184, 0), Qt.DiagCrossPattern)
10          self.pen = QPen(Qt.DashLine)
11          self.pen.setWidth(5.5)
12          self.font = QFont("Arial", 18, QFont.Medium)
13          self.text = ""
14
15      def paintEvent(self, event):
16          outline = self.rect()
17          qp = QPainter(self)
18          qp.setPen(self.pen)
19          qp.setBrush(self.brush)
20          qp.setFont(self.font)
21          qp.drawRect(outline) # draw a rectangle on the bounding box of this widget
22          text = "width, height\n(%d, %d)"
23          text = text % (outline.width(), outline.height())
24          qp.drawText(outline, Qt.AlignCenter, text) # draw the text inside the widget
25
26  class MyWin(QWidget):
27      def __init__(self):
28          super(MyWin, self).__init__()
29          self.initGUI()
30
31      def initGUI(self):
32          self.setWindowTitle("Widget Outlines")
33          height = 200
34          width = 500
35          self.setGeometry(0,0, width, height)
36          widget1 = Outline(self)
37          widget1.setGeometry(5, 5, width-10, height/2-10)
38          widget2 = Outline(self)
39          widget2.setGeometry(5, height/2, (width-20)/3, height/2-10)
40          widget3 = Outline(self)
41          widget3.setGeometry(width / 3 + 5, height/2, (width-20)/3, height/2-10)
43          widget4 = Outline(self)
44          widget4.setGeometry((width / 3)*2 + 5, height/2, (width-20)/3, height/2-10)
45          self.show()
46
47  if __name__ == '__main__':
48      app = QApplication(sys.argv)
49      mywin = MyWin()
50      sys.exit(app.exec_())
```

A few things to note in this program...

Line 2 & 3 has us importing the module `Qt` from `PyQt5.QtCore`. This module is where most of the constants in Qt are stored, including things like preset colors, pattern and line styles. In ***line 3*** we import all the classes we'll be using to draw to the canvas, all the drawing classes are found in the `PyQt5.QtGui` module.

Line 6 is the beginning of the class definition for our custom widget, `Outline`. The init method on ***Line 7-14*** creates some instance variables with the brush, pen and font styles we'll be using to draw to the canvas. I create them here instead of in the `paintEvent()` method so that they only need to be created once and can be reused.

```
self.brush = QBrush(QColor(255, 184, 0), Qt.DiagCrossPattern)
self.pen = QPen(Qt.DashLine)
self.pen.setWidth(5.5)
self.font = QFont("Arial", 18, QFont.Medium)
```

Line 15-24 is the most important bit. Here I override the `paintEvent()` to include my own custom instructions for drawing the widget. First I create a QPainter instance, then set my line, fill and font styles, and finally draw the rectangle and the text.

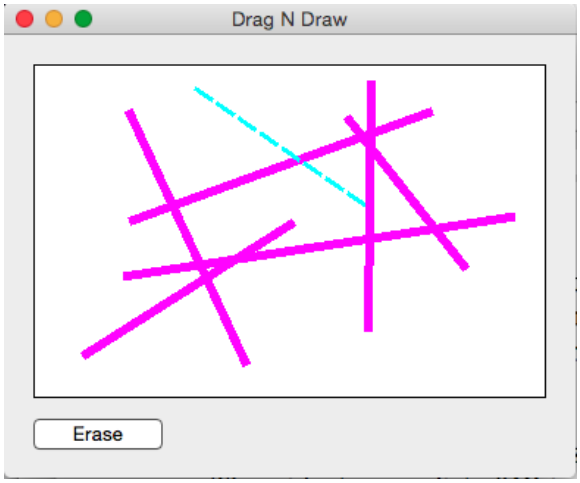
```
qp = QPainter(self)
qp.setPen(self.pen)
qp.setBrush(self.brush)
qp.setFont(self.font)
qp.drawRect(outline) # draw a rectangle on the bounding box of this widget
text = "width, height\n(%d, %d)"
text = text % (outline.width(), outline.height())
qp.drawText(outline, Qt.AlignCenter, text) # draw the text inside the widget
```

One thing worth noting is that, like most event responder methods, `paintEvent(self, event)` has an extra input argument : `event`. This is an object representing the event that triggered the event responder, and includes useful information about the event (e.g. the position of the mouse for a mouse event). The `event` argument will always a subclass of `QEvent`, in the case of a "redraw yourself" event it's an instance of `QPaintEvent`.

Line 26 and onward defines the application window (`MyWin`) and sets up four instances of `Outline` inside of it, using a bit of math to position the widgets absolutely inside the window.

Responding to Mouse Input

Your custom widgets wouldn't be very interesting if they didn't also have unique interaction paradigms. To implement user interaction with your widget you'll override the event responder functions of `QWidget` that are called in response to user input events like mouse clicks and keyboard input. Look through the documentation of `QWidget` for a full list of event responder functions, particularly the section on Events.



A line drawing widget, the current line being drawn is rendered using a different pen style than the rest

The following program creates a custom widget that allows the user to draw lines by clicking and dragging. The widget uses a list of QLine objects in order to keep track of the lines that have been drawn. Whenever the user clicks somewhere, a new QLine is created with the mouse's current location as its starting point, the line's ending point is updated every time the user moves their mouse, and the line is finalized and added to the list when the user lets go of the mouse button. An erase button is provided that empties the canvas.

Line-Canvas.py

```
1  from PyQt5.QtWidgets import *
2  from PyQt5.QtCore import Qt, QLine
3  from PyQt5.QtGui import QBrush, QPen, QColor, QFont, QPainter
4  import sys
5
6  class LineDrawCanvas(QWidget):
7      def __init__(self, parent=None):
8          super(LineDrawCanvas, self).__init__(parent)
9          self.setMinimumSize(200, 200)
10         self.linePen = QPen(Qt.SolidLine)
11         self.linePen.setColor(Qt.magenta)
12         self.linePen.setWidth(6)
13         self.drawingPen = QPen(Qt.DashLine)
14         self.drawingPen.setColor(Qt.cyan)
15         self.drawingPen.setWidth(3)
16         self.currentLine = None
17         self.lines = []
18
19     def reset(self):
20         self.lines = []
21         self.update()
22
23     def paintEvent(self, event):
24         qp = QPainter(self)
```

```
25          qp.setPen(QPen(Qt.black))
26          qp.setBrush(QBrush(Qt.white))
27          bounds = self.rect()
28          qp.drawRect(0, 0, bounds.width()-1, bounds.height()-1) # draw an outline around the graph
29          qp.setPen(self.linePen)
30          for line in self.lines:
31              qp.drawLine(line)
32          if self.currentLine != None:
33              qp.setPen(self.drawingPen)
34              qp.drawLine(self.currentLine)
35
36      def mousePressEvent(self, event):
37          self.currentLine = QLine(event.pos(), event.pos()) # start a new line
38
39      def mouseMoveEvent(self, event):
40          if self.currentLine != None:
41              self.currentLine.setP2(event.pos()) # update the second point of the line
43              self.update() # redraw the widget to show the updated line
44
45      def mouseReleaseEvent(self, event):
46          if self.currentLine != None:
47              self.lines.append(self.currentLine)
48              self.currentLine = None
49              self.update()
50
51  class MyWin(QWidget):
52      def __init__(self):
53          super(MyWin, self).__init__()
54          self.initGUI()
55
56      def initGUI(self):
57          self.setWindowTitle("Drag N Draw")
58          layout = QVBoxLayout()
59          self.setLayout(layout)
60          canvas = LineDrawCanvas()
61          resetButton = QPushButton("Erase")
62          resetButton.setMaximumWidth(100)
63          resetButton.pressed.connect(canvas.reset)
64          layout.addWidget(canvas)
65          layout.addWidget(resetButton)
66          self.setGeometry(0,0, 600, 600)
67          self.show()
68
69  if __name__ == '__main__':
70      app = QApplication(sys.argv)
71      mywin = MyWin()
72      sys.exit(app.exec_())
```

A few things to note in this program...

Take moment to understand how `LineDrawCanvas` responds visually to user input. The user input functions (mousePressEvent, mouseMoveEvent, mouseReleaseEvent) don't do any drawing directly. Instead, they update underlying variables that represent what is drawn to the screen. The `paintEvent()` method then stupidly draws those underlying representations. There is a clean separation between ***internal state*** and ***presentation of that state*** to the user. This is a common design pattern in UI programming.

In ***Lines 7-17*** the init method sets up all the instance variables of the object, including the pen styles for drawn lines and lines-in-progress, a variable to hold the current line-in-progress, and an empty list that will be filled with lines as the user draws them.

The `reset()` method on ***Lines 19-21*** simply resets the canvas by emptying out the list of lines. It also calls the `update()` method of `QWidget`, which forces the widget to redraw itself. is the slot we will connect the Erase button to.

Lines 23-34 are where the paintEvent() method is defined. The for loop on ***line 30-31*** is where the list of lines is drawn to the canvas. ***Lines 32-34*** draws the current line if the user is in the process of drawing one.

```
for line in self.lines:
    qp.drawLine(line)
if self.currentLine != None:
    qp.setPen(self.drawingPen)
    qp.drawLine(self.currentLine)
```

Lines 36-49 define the event responder functions that handle the mouse interaction for drawing lines. There are three event responder functions: one for the initial mouse press, one for mouse movement while pressed, and one for the release of the mouse. The basic

logic shouldn't be too difficult to understand: start a new line on a mouse press, update that line as the mouse moves, and add the line to the list of lines when the mouse is released. The event argument of these functions is a `QMouseEvent`, whose `pos()` method gives you a `QPoint` representing the (x,y) position of the mouse when the event happened.

Custom Signals

Signals and Slots are the main abstraction provided by Qt for connecting user interactions with widgets to the functional code of your application. Recall that you use the `connect()` method of signals to link them to other functions. For example, if you wanted a button in your app to load some recent twitter posts into your UI, you might write some code like this.

```
loadButton = QPushButton("Load Latest Posts")
loadButton.pressed.connect(self.loadLatestFeed)
```

Where `loadLatestFeed()` is a method you've defined, and `pressed` is a signal of `QPushButton` that is triggered when the user presses the button. You might now be wondering what `pressed` actually is. We know it's a signal. But how was it created and how does QPushButton trigger it?

This section will take a look at how you can create new signals that can be connected to other functions when the user interacts in specific ways with your custom widgets.

Visit the PyQt5 reference page on signals and slots for more information!
http://pyqt.sourceforge.net/Docs/PyQt5/signals_slots.html

You define new signals as class variables just after the class definition statement of your custom widget. The new signal object is created using a special function, `pyqtSignal(type1, type2, ...)`, like this:

```
class MyCoolWidget(QWidget):
    valueChanged = pyqtSignal(int)
    pressed = pyqtSignal()

    # The rest of your class code goes here...
```

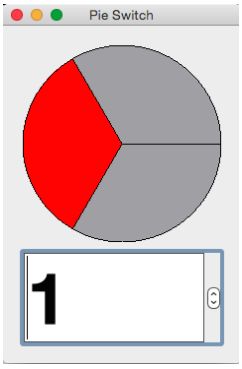
This code creates two signals for `MyCoolWidget`, `valueChanged` and `pressed`. The arguments of `pyqtSignal()` are the data types of any values that should be sent along with the signal. For example, if you have a slider-type widget, you probably want to send the latest value of the slider in a signal whenever the user changes its position. If you have a signal that sends no data, like a button press, just use `pyqtSignal()` with no arguments.

When you want to trigger a signal, use the `emit()` method of your signal like this:

```
    def mousePressEvent(self, event):
        pressed.emit()
        self.updateValue(event.pos())
        valueChanged.emit(self.value)
```

The `pressed` signal is triggered, or 'emitted', when a mouse press event is sent to your widget. A function updates the instance variable `self.value` and the `valueChanged` signal is emitted, including the updated value.

Let's look at how this works inside a complete program. The following example creates a new custom widget called `PieSwitch`, which is a three-way toggle switch represented as a pie. Whenever the user selects a new piece of the pie, the `PieSwitch` emits a signal, `changed`, that includes the index of the recently selected slice. The signal is connected to a `QSpinBox` that displays the currently selected index.



A PieSwitch with its left slice selected

PieSwitch.py

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import Qt, QObject, QPoint, QRect, pyqtSignal
from PyQt5.QtGui import QBrush, QPen, QColor, QFont, QPainter
import sys
import math

# Converts a QPoint to a point in polar coordinates and returns a tuple of (radius, theta(degrees))
def toPolar(point):
    x = point.x()
    y = point.y()
    radius = math.hypot(x,y)
    angle = math.degrees(math.atan(float(y) / x))
    if x < 0:
        angle += 180
    elif y < 0:
        angle += 360
    return radius, angle

class PieSwitch(QWidget):
    changed = pyqtSignal(int)

    def __init__(self, parent=None):
        super(PieSwitch, self).__init__(parent)
        self.selectedSlice = -1
        self.setMinimumSize(200, 200)
        self.setMaximumSize(200, 200)
        self.unselectedBrush = QBrush(Qt.gray)
        self.selectedBrush = QBrush(Qt.red)

    def setSlice(self, slicenumber):
        if slicenumber == 0 or slicenumber == 1 or slicenumber == 2:
            if self.selectedSlice != slicenumber:
                self.selectedSlice = slicenumber
                self.changed.emit(self.selectedSlice) # notify all listeners
                self.update() # redraw the widget

    def paintEvent(self, event):
        qp = QPainter(self)
        bounds = self.rect()
        qp.setBrush(self.unselectedBrush)
        if self.selectedSlice is 0:
            qp.setBrush(self.selectedBrush)
        qp.drawPie(bounds, 0, -120*16)
        qp.setBrush(self.unselectedBrush)
        if self.selectedSlice is 1:
            qp.setBrush(self.selectedBrush)
        qp.drawPie(bounds, -120*16, -120*16)
        qp.setBrush(self.unselectedBrush)
        if self.selectedSlice is 2:
            qp.setBrush(self.selectedBrush)
        qp.drawPie(bounds, -2*120*16, -120*16)


    def mousePressEvent(self, event):
        w = self.rect().width()
        h = self.rect().height()
        # Convert the x,y position of the mouse to a point on the circle
        circleCenter = QPoint(w / 2, h / 2)
        radius, angle = toPolar(event.pos() - circleCenter)
        if radius < 100:
            # The mouse is inside the circle, divide by 120 to get the pie piece index
            pie = int(angle / 120)
            self.setSlice(pie)

class MyWin(QWidget):
    def __init__(self):
        super(MyWin, self).__init__()
        self.initGUI()

    def initGUI(self):
        self.setWindowTitle("Pie Switch")
```

```
73          switch = PieSwitch()
74          spinbox = QSpinBox()
75          spinbox.setMaximum(2)
76          spinbox.setStyleSheet("QSpinBox {font-size: 72px;font-weight: bold;}")
77          switch.changed.connect(spinbox.setValue)
78          spinbox.valueChanged.connect(switch.setSlice)
79          layout = QVBoxLayout()
80          layout.addWidget(switch)
81          layout.addWidget(spinbox)
82          self.setLayout(layout)
83          self.show()
84
85  if __name__ == '__main__':
86      app = QApplication(sys.argv)
87      mywin = MyWin()
88      sys.exit(app.exec_())
```

A few things to note in this program...

There's a lot of widgets in this program, but I won't go into detail on all of them. You can do your own research by looking up each widget and its functionality online. One striking thing about this interface is that it uses a QGridLayout to align all the widgets nicely. A grid layout treats your interface as a table of rows and columns. Using the function `addWidget(row, column, rowspan, columnspan)` you can set the row and column where you want your widget to appear. You can also optionally set a rowspan and columnspan for widgets that should span over multiple rows and columns. *Note that rows and columns start counting at 0!*

Lines 8-17 is a little helper function I wrote that converts a cartesian x,y point to a radius and angle in polar coordinates. This is useful for detecting which section of the pie a mouse click occurred in.

Line 20 defines the changed signal, which sends a single integer value when emitted.

In ***lines 22-28*** the init method sets up instance variables for the widget. `selectedSlice`, which holds the index of the currently selected pie slice, and -1 if no slice is selected. And two brush styles, one for the selected slice, and one for the unselected slices.

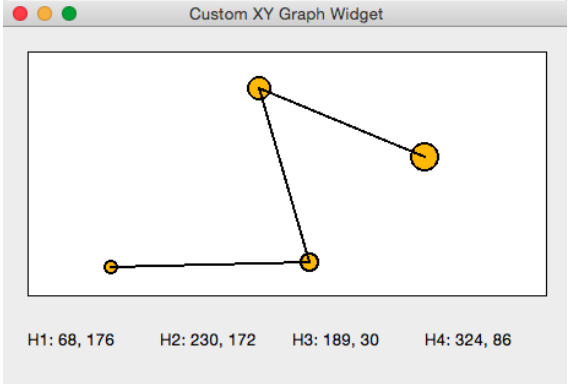
Lines 30-35 define `setSlice()`, which is the method we use to change the currently selected slice. It's used as a helper function for the user interaction behavior defined in `mousePressEvent`, and is also used as a slot on ***line 78***.

```
def setSlice(self, slicenumber):
     if slicenumber == 0 or slicenumber == 1 or slicenumber == 2:
         if self.selectedSlice != slicenumber:
             self.selectedSlice = slicenumber
             self.changed.emit(self.selectedSlice) # notify all listeners
             self.update() # redraw the widget
```

That call to `self.update()` is easy to forget, but it's important! Qt tries to be as efficient as possible and only asks widgets to redraw themselves when absolutely necessary. Most of the time its up to you to tell your widgets to redraw themselves when something has changed in the underlying state variables.

XY Breakpoint Editor Widget

Let's look at one last example that brings together all of these ideas. This program creates a breakpoint editor widget. A breakpoint editor is an XY line graph with a number of "handles" that can be dragged around. This kind of widget is sometimes used in audio production software to set filter parameters, but can be used for all kinds of things. Our breakpoint editor will have four draggable handles, whose positions are reflected below the editor on a number of labels. To represent the handles in our program we'll make a new `Handle` class. Each handle will store its own position and have a custom signal that's emitted whenever its position is changed.



The breakpoint editor widget we'll build in this example

To make displaying the positions of the handles easier we'll make a subclass of `QLabel`, `HandleLabel`, and give it a method for updating itself directly from a `Handle` instance.

Point-Dragger.py

```
from PyQt5.QtWidgets import *
from PyQt5.QtCore import Qt, QObject, QPoint, QRect, pyqtSignal
from PyQt5.QtGui import QBrush, QPen, QColor, QFont, QPainter
import sys

class Handle(QObject):
    handleMoved = pyqtSignal(QPoint)

    def __init__(self, label, xpos, ypos, radius):
        super(Handle, self).__init__()
        self.pos = QPoint(xpos, ypos)
        self.label = label
        self.radius = radius

    def setPos(self, newpos):
        self.pos.setX(newpos.x())
        self.pos.setY(newpos.y())
        self.handleMoved.emit(self.pos)

    def x(self):
        return self.pos.x()

    def y(self):
        return self.pos.y()

    def bounds(self):
        return QRect(self.x()-self.radius, self.y()-self.radius, self.radius * 2, self.radius * 2)

class XYGraph(QWidget):
    def __init__(self, parent=None):
        super(XYGraph, self).__init__(parent)
        self.handleBrush = QBrush(QColor(255, 184, 0))
        self.backgroundBrush = QBrush(Qt.white)
        self.linePen = QPen(Qt.SolidLine)
        self.linePen.setWidth(2)
        self.currentHandle = None
        self.dragStartPos = None
        self.setMinimumSize(400, 200)
        self.handles = [Handle('H1',70,80,5), Handle('H2',140,130,7), Handle('H3',270,60,9),
Handle('H4',350,150,11)]

    def paintEvent(self, event):
        qp = QPainter(self)
        qp.setPen(self.linePen)
        qp.setBrush(self.backgroundBrush)
        qp.drawRect(self.rect()) # draw an outline around the graph
        qp.setBrush(self.handleBrush)
        lasthandle = None
        for handle in self.handles:
            qp.drawEllipse(handle.bounds())
            if lasthandle != None:
                qp.drawLine(lasthandle.x(), lasthandle.y(), handle.x(), handle.y())
            lasthandle = handle

    def mousePressEvent(self, event):
        mouseX = event.pos().x()
        mouseY = event.pos().y()
        self.currentHandle = None
        for handle in self.handles:       # Check if the mouse is inside the bounds of one of the handles
            h = handle.bounds()
            if mouseX > h.left() and mouseX < h.right() and mouseY > h.top() and mouseY < h.bottom():
                # mouse is inside this handle's bounding box
                self.currentHandle = handle
                self.dragStartPos = QPoint(handle.x(), handle.y())
                break

    def mouseMoveEvent(self, event):
        if self.currentHandle != None:
            w = self.width()
            h = self.height()
            mouse = event.pos()
```

```
 73              if  mouse.x() > 0 and mouse.x() < w and mouse.y() > 0 and mouse.y() < h:
 74                  self.currentHandle.setPos(mouse)
 75              else:
 76                  self.currentHandle.setPos(self.dragStartPos)
 77                  self.dragStartPos = None
 78                  self.currentHandle = None
 79              self.update()
 80
 81      def mouseReleaseEvent(self, event):
 82          if self.currentHandle != None:
 83              self.currentHandle = None
 84
 85  class HandleLabel(QLabel):
 86      def __init__(self, handle):
 87          super(HandleLabel, self).__init__()
 88          self.label = handle.label
 89          self.updateLabel(handle.pos)
 90          self.setMinimumSize(100, 50)
 91          handle.handleMoved.connect(self.updateLabel)
 92
 93      def updateLabel(self, handle):
 94          text = "%s: %d, %d" % (self.label, handle.x(), handle.y())
 95          self.setText(text)
 96
 97  class MyWin(QWidget):
 98      def __init__(self):
 99          super(MyWin, self).__init__()
100          self.initGUI()
101
102      def initGUI(self):
103          mainlayout = QVBoxLayout()
104          sublayout = QHBoxLayout()
105          graph = XYGraph()
106          for handle in graph.handles:                    # Make label widgets for all the handles
107              sublayout.addWidget(HandleLabel(handle))
108          mainlayout.addWidget(graph)
109          mainlayout.addLayout(sublayout)
110          self.setLayout(mainlayout)
111          self.setWindowTitle("Custom XY Graph Widget")
112          self.show()
113
114  if __name__ == '__main__':
115      app = QApplication(sys.argv)
116      mywin = MyWin()
117      sys.exit(app.exec_())
```

A few things to note in this program...

Lines 6-27 define the `Handle` class that is used as an underlying representation of each handle in the `XYGraph` widget. Each `Handle` stores its own position, its label and the radius of its circle as instance variables.

Notice that `Handle` extends `QObject`, not `QWidget`. `Handle` isn't meant to be used as a widget, but purely as a data structure that represents a handle in the GUI. By extending `QObject` we can still give Handle access the slots and signals functionality of Qt without all the extra unnecessary functionality of `QWidget`. On ***line 7*** I immediately make use of this functionality by giving `Handle` a signal, `handleMoved`, which is emitted every time the handle's x,y position is changed. `handleMoved` sends a `QPoint` containing the latest position of the handle.

Lines 15-18 are where the `setPos` method of handle is defined, which takes as input `newpos`, a `QPoint` representing the new position of the handle. This is the method used in other code to change the handle's position values, it's also where the `handleMoved` signal is emitted.

```
    def setPos(self, newpos):
        self.pos.setX(newpos.x())
        self.pos.setY(newpos.y())
        self.handleMoved.emit(self.pos)
```

The definition for the `XYGraph` class starts on ***line 29***, with the init method on ***lines 30-39*** setting up some important instance variables, such as `currentHandle` and `dragStartPos` which are used to keep track of mouse interaction, and `handles` which is a list containing the four handle objects that represent what will be drawn in `XYGraph`'s `paintEvent` method.

```
self.handles = [Handle('H1',70,80,5), Handle('H2',140,130,7), Handle('H3',270,60,9), Handle('H4',350,150,11)]
```

Lines 50-54 inside the `paintEvent` method are where all the drawing happens. We have a for loop that iterates over the list of handle objects, drawing an ellipse for each and a line connecting the current handle to the one that came before.

```
        for handle in self.handles:
            qp.drawEllipse(handle.bounds())
            if lasthandle != None:
                qp.drawLine(lasthandle.x(), lasthandle.y(), handle.x(), handle.y())
            lasthandle = handle
```

The methods `mousePressEvent`, `mouseMoveEvent` and `mouseReleaseEvent` on ***lines 56-83*** together handle user interaction with the graph. When a mouse press happens, the list of handles are iterated over in search of a handle whose bounds contain the current mouse position. If one is found, that handle becomes the current handle, and the starting position of the dragging operation is stored. We store this so that we can return the handle to its original location if the user cancels this dragging operation by moving their mouse outside the bounds of the widget.

The following excerpt from `mouseMoveEvent` (***line 73-79***) is responsible for checking if the user has moved outside the bounds of the widget, in which case, the current handle is returned to its starting position, and the whole drag operation is reset.

```
            if  mouse.x() > 0 and mouse.x() < w and mouse.y() > 0 and mouse.y() < h:
                self.currentHandle.setPos(mouse)
            else:
                self.currentHandle.setPos(self.dragStartPos)
                self.dragStartPos = None
                self.currentHandle = None
            self.update()
```

Lines 85-95 creates a custom widget called `HandleLabel`, which is just a `QLabel` with a few extra bits of functionality that make it easier to work with our `Handle` objects. Importantly, the `init` method of our new label takes as its input the `Handle` instance whose data it will display. On ***line 91*** the `handleMoved` signal of the handle it will display is connected to its own `updateLabel` method, which takes care of converting the label and position of the handle into a text string that can be shown to the user.

```
class HandleLabel(QLabel):
    def __init__(self, handle):
        super(HandleLabel, self).__init__()
        self.label = handle.label
        self.updateLabel(handle.pos)
        self.setMinimumSize(100, 50)
        handle.handleMoved.connect(self.updateLabel)

    def updateLabel(self, handle):
        text = "%s: %d, %d" % (self.label, handle.x(), handle.y())
        self.setText(text)
```

Exercises

Take some time to study and understand the above examples. If you're wondering what the inputs to a method are, or what a certain Qt class does, look them up in the PyQt5 documentation.

As a further exercise, try to modify the XYGraph program to draw a fourth line connecting the last two handles, so that it draws an adjustable quadralateral shape.

Next try to add another handle so that the graph displays 5 adjustable handles. After that see if you can modify the paintEvent method of XYGraph to connect the handles with curves instead of lines.

Once you're satisfied, think about how custom widgets could play into your own interface. Do you need to make custom widgets, with unique interaction paradigms, or are the default widgets provided by Qt enough?

I strongly suggest you study the Qt stylesheet reference page.
This reference includes a full list of the widgets that can be styled and their properties.

http://doc.qt.io/qt-5/stylesheet-reference.html
