PySide/PyQt Tutorial: Creating Your Own Signals and Slots

This article is part 5 of 8 in the series Python PySide/PyQt Tutorial

You don't have to rely solely on the signals that are provided by Qt widgets, however; you can create your own. Signals are created using the Signal class. A simple signal definition would be:

PySide

```
1 from PySide.QtCore import Signal
2 tapped = Signal()
```

PyQt

```
1 from PyQt4.QtCore import pyqtSignal
2 tapped = pyqtSignal()
```

Then, when the conditions for the object being tapped are satisfied, you call the signal's `emit` method, and the signal is emitted, calling any slots to which it is connected:

```
1 thing.tapped.emit()
```

This is good for two reasons; first, it allows users of your objects to interact with them in familiar ways; and second, it allows your objects to be used more flexibly, leaving the definition effects of actions on your object to the code that uses them.

A Simple PySide/PyQt Signal Emitting Example

Let's define a simple `PunchingBag` class that does only one thing: when its `punch` is called, it emits a `punched` signal:

PySide

```
1  from PySide.QtCore import QObject, Signal, Slot
2
3  class PunchingBag(QObject):
4      ''' Represents a punching bag; when you punch it, it
5          emits a signal that indicates that it was punched. '''
6      punched = Signal()
7
8      def __init__(self):
9          # Initialize the PunchingBag as a QObject
10         QObject.__init__(self)
11
12     def punch(self):
13         ''' Punch the bag '''
14         self.punched.emit()
```

PyQt

```
1  from PyQt4.QtCore import QObject, pyqtSignal, pyqtSlot
2
3  class PunchingBag(QObject):
4      ''' Represents a punching bag; when you punch it, it
5          emits a signal that indicates that it was punched. '''
6      punched = pyqtSignal()
```

```
7
8       def __init__(self):
9           # Initialize the PunchingBag as a QObject
10          QObject.__init__(self)
11
12      def punch(self):
13          ''' Punch the bag '''
14          self.punched.emit()
```

You can easily see what we've done. The `PunchingBag` inherits from `QObject` so it can emit signals; it has a signal called `punched`, which carries no data; and it has a `punch` method which does nothing but emit the `punched` signal.

To make our `PunchingBag` useful, we need to connect its `punched` signal to a slot that does something. We'll define a simple one that prints, "Bag was punched" to the console, instantiate our `PunchingBag`, and connect its `punched` signal to the slot:

PySide

```
1 @Slot()
2 def say_punched():
3     ''' Give evidence that a bag was punched. '''
4     print('Bag was punched.')
5
6 bag = PunchingBag()
7 # Connect the bag's punched signal to the say_punched slot
8 bag.punched.connect(say_punched)
```

PyQt

```
1 @pyqtSlot()
2 def say_punched():
3     ''' Give evidence that a bag was punched. '''
4     print('Bag was punched.')
5
6 bag = PunchingBag()
7 # Connect the bag's punched signal to the say_punched slot
8 bag.punched.connect(say_punched)
```

Then, we'll punch the bag and see what happens:

```
1 # Punch the bag 10 times
2 for i in range(10):
3     bag.punch()
```

When you put it all in a script and run it, it will print:

```
1  Bag was punched.
2  Bag was punched.
3  Bag was punched.
4  Bag was punched.
5  Bag was punched.
6  Bag was punched.
7  Bag was punched.
8  Bag was punched.
9  Bag was punched.
10 Bag was punched.
```

Effective, but not particularly impressive. However, you can see the usefulness of it: our punching bag would be a good fit anywhere you need a bag that reacts to punching, because the `PunchingBag` leaves implementation of a reaction to punching to the code that uses it.

Data-Carrying PySide/PyQt Signals

One of the most interesting things you can do when creating signals is to make them carry data. For example, you could make a signal carry an integer, thus:

PySide

```
1 updated = Signal(int)
```

PyQt

```
1 updated = pyqtSignal(int)
```

or a string:

PySide

```
1 updated = Signal(str)
```

PyQt

```
1 updated = pyqtSignal(str)
```

The datatype may be any Python type name or a string identifying a C++ datatype. Since this tutorial presupposes no C++ knowledge, we'll stick to Python types.

A PySide/PyQt Signal-Sending Circle

Let's define a Circle with properties `x`, `y`, and `r`, denoting the `x` and `y` position of the center of the circle, and its radius, respectively. You might want to have one signal that is emitted when the circle is resized, and another that is emitted when it is moved; we'll call them `resized` and `moved`, respectively.

It would be possible to have the slots to which the `resized` and `moved` signals are connected check the new position or size of the circle and respond accordingly, but it's more convenient and requires less knowledge of circles by the slot functions if the signal that is sent can include that information.

PySide

```
1  from PySide.QtCore import QObject, Signal, Slot
2
3  class Circle(QObject):
4      ''' Represents a circle defined by the x and y
5          coordinates of its center and its radius r. '''
6      # Signal emitted when the circle is resized,
7      # carrying its integer radius
8      resized = Signal(int)
9      # Signal emitted when the circle is moved, carrying
10     # the x and y coordinates of its center.
11     moved = Signal(int, int)
12
13     def __init__(self, x, y, r):
14         # Initialize the Circle as a QObject so it can emit signals
15         QObject.__init__(self)
16
17         # "Hide" the values and expose them via properties
```

```
        self._x = x
        self._y = y
        self._r = r

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, new_x):
        self._x = new_x
        # After the center is moved, emit the
        # moved signal with the new coordinates
        self.moved.emit(new_x, self.y)

    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, new_y):
        self._y = new_y
        # After the center is moved, emit the moved
        # signal with the new coordinates
        self.moved.emit(self.x, new_y)

    @property
    def r(self):
        return self._r

    @r.setter
    def r(self, new_r):
        self._r = new_r
        # After the radius is changed, emit the
        # resized signal with the new radius
        self.resized.emit(new_r)
```

PyQt

```
from PyQt4.QtCore import QObject, pyqtSignal, pyqtSlot

class Circle(QObject):
    ''' Represents a circle defined by the x and y
        coordinates of its center and its radius r. '''
    # Signal emitted when the circle is resized,
    # carrying its integer radius
    resized = pyqtSignal(int)
    # Signal emitted when the circle is moved, carrying
    # the x and y coordinates of its center.
    moved = pyqtSignal(int, int)

    def __init__(self, x, y, r):
        # Initialize the Circle as a QObject so it can emit signals
        QObject.__init__(self)

        # "Hide" the values and expose them via properties
        self._x = x
        self._y = y
        self._r = r

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, new_x):
        self._x = new_x
        # After the center is moved, emit the
        # moved signal with the new coordinates
        self.moved.emit(new_x, self.y)

    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, new_y):
        self._y = new_y
        # After the center is moved, emit the moved
        # signal with the new coordinates
        self.moved.emit(self.x, new_y)

    @property
    def r(self):
```

```
45          return self._r
46
47      @r.setter
48      def r(self, new_r):
49          self._r = new_r
50          # After the radius is changed, emit the
51          # resized signal with the new radius
52          self.resized.emit(new_r)
```

Note these salient points:

- The `Circle` inherits from `QObject` so it can emit signals.
- The signals are created with the signature of the slot to which they will be connected.
- The same signal can be emitted in multiple places.

Now, let's define some slots that can be connected to the Circle's signals. Remember last time, when we said we'd see more about the `@Slot` decorator? We now have signals that carry data, so we'll see how to make slots that can receive it. To make a slot accept data from a signal, we simply define it with the same signature as its signal:

PySide

```
# A slot for the "moved" signal, accepting the x and y coordinates
@Slot(int, int)
def on_moved(x, y):
    print('Circle was moved to (%s, %s).' % (x, y))

# A slot for the "resized" signal, accepting the radius
@Slot(int)
def on_resized(r):
    print('Circle was resized to radius %s.' % r)
```

PyQt

```
# A slot for the "moved" signal, accepting the x and y coordinates
@pyqtSlot(int, int)
def on_moved(x, y):
    print('Circle was moved to (%s, %s).' % (x, y))

# A slot for the "resized" signal, accepting the radius
@pyqtSlot(int)
def on_resized(r):
    print('Circle was resized to radius %s.' % r)
```

Very simple and intuitive. For more information on Python decorators, you might want to checkout the article - Python Decorators Overview to familiarise yourself.

Finally, let's instantiate a Circle, hook up the signals to the slots, and move and resize it:

```
c = Circle(5, 5, 4)

# Connect the Circle's signals to our simple slots
c.moved.connect(on_moved)
c.resized.connect(on_resized)

# Move the circle one unit to the right
c.x += 1

# Increase the circle's radius by one unit
c.r += 1
```

When you run the resulting script, your output should be:

```
Circle was moved to (6, 5).
Circle was resized to radius 5.
```

Now that we've developed a better understanding of signals and slots, we are ready to use some more advanced widgets. In our next instalment, we will begin to discuss the `QListWidget` and `QListView`, two ways of creating list box controls.