Table of content

Selectable list of Python objects in QML

This PySide code example shows you how to display a list of arbitrary Python objects in QML and to get the "real" Python object in a callback when the user clicks on a list item. This is done by wrapping the Python objects inside a QObject. Of course, if you are developing a new application, you can create your model entity objects directly as QObject subclasses, but for adding a QML UI on top of existing Python code, this should be very useful.

This example consists of two files:

* PythonList.py – Our Python code
* PythonList.qml – The corresponding QML UI

PythonList.py

This is the main module. After finishing writing it, you can start it using python PythonList.py

Import the required modules

This is pretty straightforward. If you don't have OpenGL support on your target platform (or you don't want to use OpenGL-accelerated QML), simply remove the line from PySide import QtOpenGL.

```
import sys

from PySide import QtCore
from PySide import QtGui
from PySide import QtDeclarative
from PySide import QtOpenGL
```

Define a simple QObject wrapper

Create a new QObject subclass that gets your custom Python object (called "thing" in this example) as constructor parameter. You can then define different properties (QtCore.Property) for all the parts of the thing that you want to show in the UI. You will use these property names to access attributes of your object in QML later.

```
class ThingWrapper(QtCore.QObject):
    def __init__(self, thing):
        QtCore.QObject.__init__(self)
        self._thing = thing

    def _name(self):
        return str(self._thing)

    changed = QtCore.Signal()

    name = QtCore.Property(unicode, _name, notify=changed)
```

Define your list model for your objects

You can subclass QAbstractListModel and implement the required methods to provide a proper data model to QML's ListView. Theoretically, you could have multiple columns, but for simplicity, we just use one here, and then access the different attributes of our objects (one object per "row"). The list model here gets a list of (wrapped) "things" as constructor parameter. You could also generate objects on the fly if you want, just make sure that you return the proper value in the rowCount function.

```
class ThingListModel(QtCore.QAbstractListModel):
    COLUMNS = ('thing',)

    def __init__(self, things):
        QtCore.QAbstractListModel.__init__(self)
        self._things = things
        self.setRoleNames(dict(enumerate(ThingListModel.COLUMNS)))

    def rowCount(self, parent=QtCore.QModelIndex()):
        return len(self._things)

    def data(self, index, role):
        if index.isValid() and role == ThingListModel.COLUMNS.index('thing'):
            return self._things[index.row()]
        return None
```

Create a controller for receiving events

For receiving events from QML, there are several possiblities. We simply create another QObject subclass and give it a Slot with one parameter (the wrapper object in the selected row). We pass the wrapper object from QML, and arrive at this point in the “Python world” and can do whatever we want with the object the user has selected.

```
class Controller(QtCore.QObject):
    @QtCore.Slot(QtCore.QObject)
    def thingSelected(self, wrapper):
        print 'User clicked on:', wrapper._thing.name
        if wrapper._thing.number > 10:
            print 'The number is greater than ten!'
```

Set up the QDeclarativeView

Here follows some generic boilerplate code for setting up the basic QApplication, QMainWindow and QDeclarativeView object tree to display QML. If you don’t want to use OpenGL acceleration, remove the lines glw = QtOpenGL.QGLWidget() and view.setViewport(glw).

```
app = QtGui.QApplication(sys.argv)

m = QtGui.QMainWindow()

view = QtDeclarative.QDeclarativeView()
glw = QtOpenGL.QGLWidget()
view.setViewport(glw)
view.setResizeMode(QtDeclarative.QDeclarativeView.SizeRootObjectToView)
```

Your custom Python object

In an existing project, you would simply import your data model module and be done with it. Here, we create a dummy Person object (note that it is a pure Python object and does not know anything about QObject or PySide!) and some example data:

```
class Person(object):
    def __init__(self, name, number):
        self.name = name
        self.number = number
    def __str__(self):
        return 'Person "%s" (%d)' % (self.name, self.number)

people = [
        Person('Locke', 4),
        Person('Reyes', 8),
        Person('Ford', 15),
        Person('Jarrah', 16),
        Person('Shephard', 23),
        Person('Kwon', 42),
]
```

Wrap your custom objects in QObjects

As we have defined our ThingWrapper class above, this is pretty straightforward using Python's list comprehensions:

```
things = [ThingWrapper(thing) for thing in people]
```

Connect the controller, the model and load the QML file

Again, this is easy, as we have already written most of the code and just need to glue all the parts together. Using setContextProperty we can expose QObject instances to the QML engine and access them using the name given as first parameter:

```
controller = Controller()
thingList = ThingListModel(things)

rc = view.rootContext()

rc.setContextProperty('controller', controller)
rc.setContextProperty('pythonListModel', thingList)

view.setSource('PythonList.qml')
```

Show the window and start the application

Add the view to the window, show the window and finally start the application:

```
m.setCentralWidget(view)

m.show()

app.exec_()
```

PythonList.qml

QML is pretty compact, and we only need to define the look of one row, and QML will take care of rendering every row separately. The important parts here are:

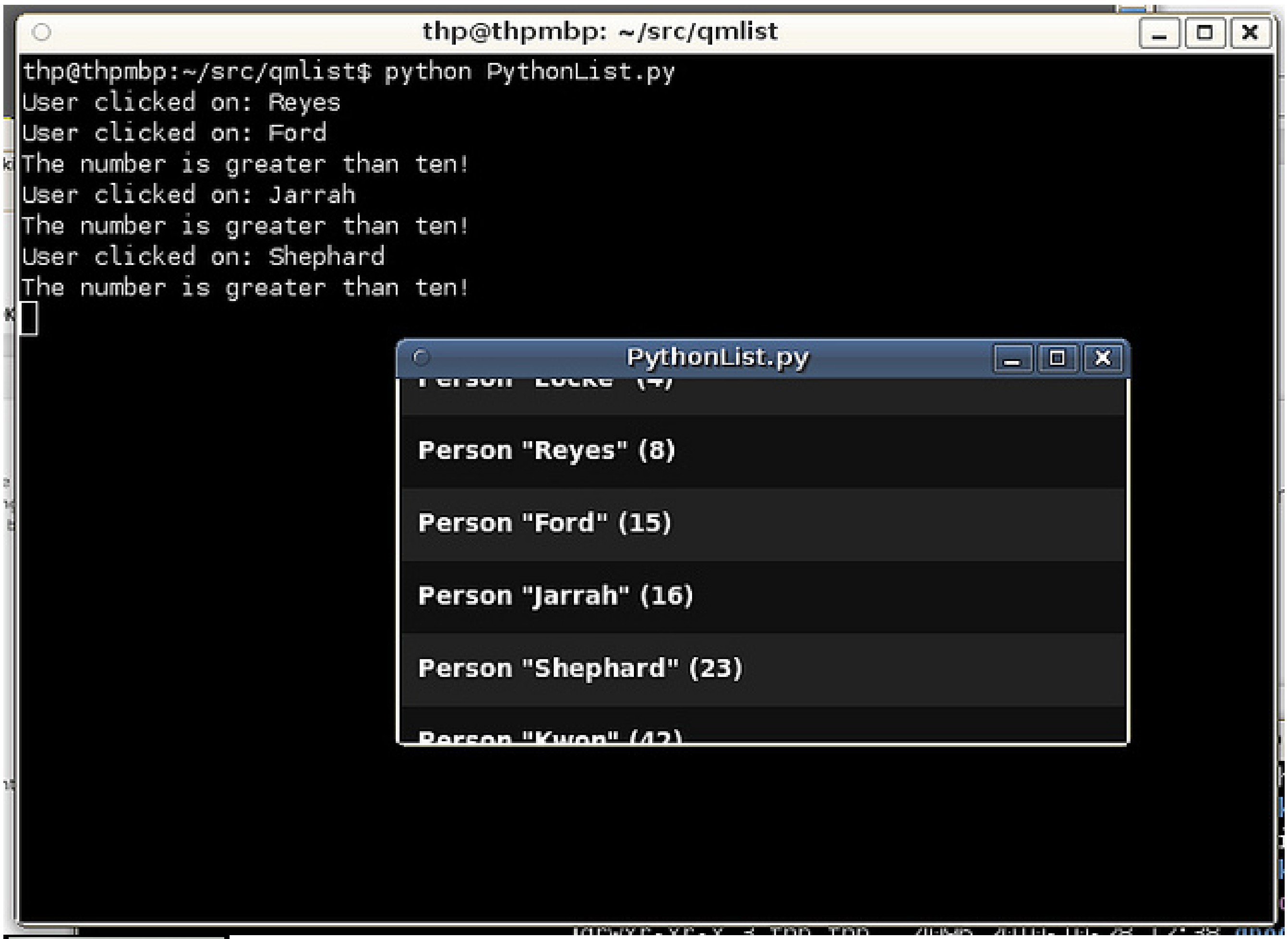
* model references the list model, the pythonListModel corresponds to the model set using setContextProperty in Python
* A nifty way to have alternating background colors: color: ((index % 2 == 0) ? ”#222” : ”#111”)
* The text attribute of the Text item is taken from model.thing.name – where model is our model, thing is the column of our model, and name is the property of our wrapper
* When the item is clicked (MouseArea), the controller (from setContextProperty) gets its

thingSelected slot called with model.thing as the first and only parameter

```
import Qt 4.7

ListView {
    id: pythonList
    width: 400
    height: 200

    model: pythonListModel

    delegate: Component {
        Rectangle {
            width: pythonList.width
            height: 40
            color: ((index % 2 == 0)?"#222":"#111")
            Text {
                id: title
                elide: Text.ElideRight
                text: model.thing.name
                color: "white"
                font.bold: true
                anchors.leftMargin: 10
                anchors.fill: parent
                verticalAlignment: Text.AlignVCenter
            }
            MouseArea {
                anchors.fill: parent
                onClicked: { controller.thingSelected(model.thing) }
            }
        }
    }
}
```

Starting the example

This is how it looks when you run the example using python PythonList.py:


```
thp@thpmbp:~/src/qmlist$ python PythonList.py
User clicked on: Reyes
User clicked on: Ford
The number is greater than ten!
User clicked on: Jarrah
The number is greater than ten!
User clicked on: Shephard
The number is greater than ten!
```