Table of content

Multi-selection lists in Python with QML

This is an extension of the Selectable list of Python objects in QML and focuses more on modifying the list and getting selected items out of the QML view.

MultiList.py

This is the Python code that takes care of setting up the model and controlling it.

Encoding declaration and module docstring

We can use the docstring as window title later on (using doc):

```
# -*- coding: utf-8 -*-

"""Click-your-Zen of PySide"""
```

Imports

We need sys for the command line arguments and we need this for some free sample data:

```
import sys
import this

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

QObject wrapper with “checked” property

Just as with the previous example, we now define a wrapper object. The difference here is that it has a “checked” property and a convenience method for toggling the property (don’t forget that you need to emit the “changed” signal, otherwise QML would not know that something has changed!).

```
class ZenWrapper(QtCore.QObject):
    def __init__(self, zenItem):
        QtCore.QObject.__init__(self)
        self._zenItem = zenItem
        self._checked = False

    def _name(self):
        return self._zenItem

    def is_checked(self):
        return self._checked

    def toggle_checked(self):
        self._checked = not self._checked
        self.changed.emit()

    changed = QtCore.Signal()

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

List model, with helper to get checked items

This is again mostly the same as in the previous example, but we add a new helper method checked that takes care of retrieving all the checked items from the zenItems list:

```
class ZenListModel(QtCore.QAbstractListModel):
    def __init__(self, zenItems):
        QtCore.QAbstractListModel.__init__(self)
        self._zenItems = zenItems
        self.setRoleNames({0: 'zenItem'})

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

    def checked(self):
        return [x for x in self._zenItems if x.checked]

    def data(self, index, role):
        if index.isValid() and role == 0:
            return self._zenItems[index.row()]
```

Controller for toggling and retrieving changes

This controller provides a toggled slot that will be called by QML when we click on an item. We toggle the checked state of the item and (for the sake of demonstration) get the list of currently-checked items and print it on the console. We also set the window title to show the amount of currently selected items:

```
class Controller(QtCore.QObject):
    @QtCore.Slot(QtCore.QObject, QtCore.QObject)
    def toggled(self, model, wrapper):
        global view, __doc__
        wrapper.toggle_checked()
        new_list = model.checked()
        print '='*20, 'New List', '='*20
        print '\n'.join(x.name for x in new_list)
        view.setWindowTitle('%s (%d)' % (__doc__, len(new_list)))
```

Model and controller setup

Here we generate some example data and instantiate the controller and the list model:

```
zenItems = [ZenWrapper(zenItem) for zenItem in \
        ''.join([this.d.get(c, c) for c in \
        this.s]).splitlines()[2:]]

controller = Controller()
zenItemList = ZenListModel(zenItems)
```

QApplication / QDeclarativeView + Glue code

This piece of code creates a new QApplication instance, a new QDeclarativeView and also exposes the controller and the list model to the QML environment, so it can be accessed from there:

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

view = QtDeclarative.QDeclarativeView()
view.setWindowTitle(__doc__)
view.setResizeMode(QtDeclarative.QDeclarativeView.SizeRootObjectToView)

rc = view.rootContext()

rc.setContextProperty('controller', controller)
rc.setContextProperty('pythonListModel', zenItemList)
view.setSource(__file__.replace('.py', '.qml'))

view.show()
app.exec_()
```

MultiList.qml

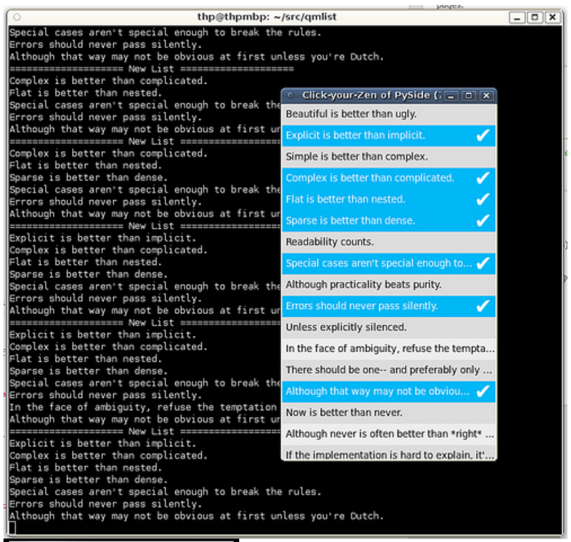
```
import Qt 4.7

ListView {
    id: pythonList
    width: 300
    height: 500
    model: pythonListModel

    delegate: Component {
        Rectangle {
            width: pythonList.width
            height: 30
            color: model.zenItem.checked?"#00B8F5":(index%2?"#eee":"#ddd")
            Text {
                elide: Text.ElideRight
                text: model.zenItem.name
                color: (model.zenItem.checked?"white":"black")
                anchors {
                    verticalCenter: parent.verticalCenter
                    left: parent.left
                    right: (model.zenItem.checked?checkbox.left:parent.right)
                    leftMargin: 5
                }
            }
            Text {
                id: checkbox
                text: " "
                font.pixelSize: parent.height
                font.bold: true
                visible: model.zenItem.checked
                color: "white"
                anchors {
                    verticalCenter: parent.verticalCenter
                    right: parent.right
                    rightMargin: 5
                }
            }
            MouseArea {
                anchors.fill: parent
                onClicked: { controller.toggled(pythonListModel, model.zenItem) }
            }
        }
    }
}
```

How the example app looks like

Starting the app using python MultiList.py should give something like this:



```
thp@thpmbp: ~/src/qmlist
Special cases aren't special enough to break the rules.
Errors should never pass silently.
Although that way may not be obvious at first unless you're Dutch.
==================== New List ====================
Complex is better than complicated.
Flat is better than nested.
Special cases aren't special enough to break the
Errors should never pass silently.
Although that way may not be obvious at first ur
==================== New List ===================
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Special cases aren't special enough to break the
Errors should never pass silently.
Although that way may not be obvious at first ur
==================== New List ===================
Explicit is better than implicit.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Special cases aren't special enough to break the
Errors should never pass silently.
Although that way may not be obvious at first ur
==================== New List ===================
Explicit is better than implicit.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Special cases aren't special enough to break the
Errors should never pass silently.
In the face of ambiguity, refuse the temptation
Although that way may not be obvious at first ur
==================== New List ===================
Explicit is better than implicit.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Special cases aren't special enough to break the rules.
Errors should never pass silently.
Although that way may not be obvious at first unless you're Dutch.
```

Click-your-Zen of PySide (:

- Beautiful is better than ugly.
- Explicit is better than implicit. ✓
- Simple is better than complex.
- Complex is better than complicated. ✓
- Flat is better than nested. ✓
- Sparse is better than dense. ✓
- Readability counts.
- Special cases aren't special enough to... ✓
- Although practicality beats purity.
- Errors should never pass silently. ✓
- Unless explicitly silenced.
- In the face of ambiguity, refuse the tempta...
- There should be one-- and preferably only ...
- Although that way may not be obviou... ✓
- Now is better than never.
- Although never is often better than *right* ...
- If the implementation is hard to explain, it'...

Content is available under Creative Commons Attribution-ShareAlike 2.5 Generic