Table of content

Defining and using constants from PySide in QML

This PySide tutorial shows you how to define constant values (with hierarchical structures) and expose them to QML. This allows you to define often-used constants (e.g. the size of a finger-friendly widget) and change them globally in your UI simply by changing the code in your Python script. Of course, as it is a simple Python Dictionary, you can also generate the data programatically. Be aware that this example does not allow you to change the values once you have set the context property (for this, you would need a QObject with a NOTIFYable properties).

Constants.py

Importing the required modules

We need the sys module for command line parameters and to give the correct exit status. We also need QtCore, QtGui and QtDeclarative to set up our UI:

```
import sys

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

Defining the constants as Python dict

Simply create a dictionary – it should contain basic data types (e.g. str, float, int, bool), dictionaries (dict) or lists (list). You can nest lists and dicts:

```
Constants = {
    'CustomText': "Hey PySide!",
    'FontSize': 9.24,
    'Colors': {
        'Background': "#8ac852",
        'Foreground': "#00672a",
    },
    'BoldText': True,
    'Items': {
        'Count': 7,
        'Suffixes': ['A', 'B', 'C', 'D', 'E', 'F', 'G'],
    },
    'Step': { 'X': 5, 'Y': 10 },
}
```

Creating QApplication and QDeclarativeView

This is easy – simply create a new QApplication, passing the command line parameters to its constructor. Then create a QDeclarativeView and configure it so that whenever the window is resized, the root object automatically changes size as well.

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

view = QtDeclarative.QDeclarativeView()
view.setResizeMode(QtDeclarative.QDeclarativeView.SizeRootObjectToView)
```

Inject the constants as context property
Get the root context of the QML engine via rootContext(), then use setContextProperty to expose the constants dict to the QML world:

```
ctx = view.rootContext()
ctx.setContextProperty('C', Constants)
```

Load QML, show window and run application
Assuming the QML file lies in the current directory, simply set its filename with setSource on the view. Then, let the window appear with show() and finally start the application using exec_() on our QApplication instance.

```
view.setSource('Constants.qml')
view.show()

sys.exit(app.exec_())
```

Constants.qml
Now that you have "injected" your constants as "C" context property, you can now access its items as if they were attributes. This also works for nested dictionaries (e.g. C.Items.Count) and also for lists (e.g. C.Items.Suffixes[index]). Be aware that with this approach, you cannot change constants later (e.g. when you want to change the background color at runtime or something.

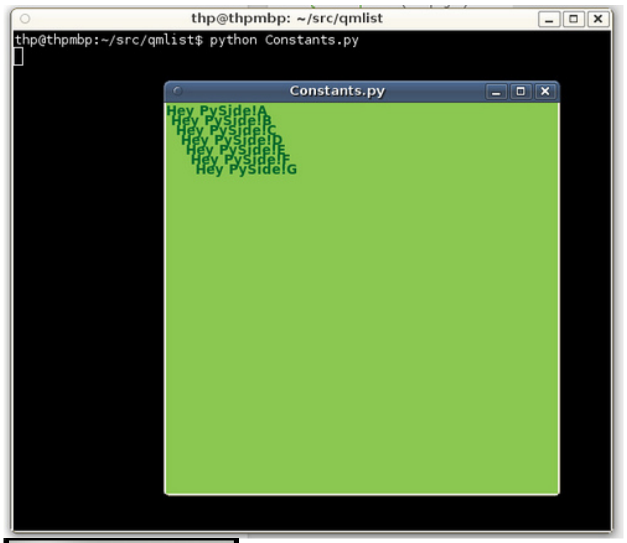
```
import Qt 4.7

Rectangle {
    width: 400
    height: 400
    color: C.Colors.Background

    Repeater {
        model: C.Items.Count

        Text {
            y: index * C.Step.Y
            x: index * C.Step.X
            color: C.Colors.Foreground
            font.bold: C.BoldText
            font.pointSize: C.FontSize
            text: C.CustomText + C.Items.Suffixes[index]
        }
    }
}
```

How the example app looks like
Start the app using python Constants.py. The result should look similar to this:



```
thp@thpmbp:~/src/qmlist$ python Constants.py
```

```
Hey PySide!A
 Hey PySide!B
  Hey PySide!C
   Hey PySide!D
    Hey PySide!E
     Hey PySide!F
      Hey PySide!G
```