Introducing PyQt4*

for GUI Application Development

David Boddie
dboddie@trolltech.com

Torsten Marek
shlomme@gmx.net

EuroPython 2006, CERN, Geneva

© 2006 David Boddie and Torsten Marek
* Qt and Trolltech are registered trademarks of Trolltech ASA

What is Qt?

Qt is a ***cross-platform C++ framework*** for the development of GUI applications.

- Developed by Trolltech in Norway and Germany
- Supported on Windows®, Mac OS X®, Linux® and other Unix® platforms
- Available under the GNU General Public License on all supported platforms
- Also available under a Commercial License for closed source applications
- Not just a widget toolkit – other features for application developers

Features

- Widgets, layouts, styles (native appearance on each platform)
- Standard GUI features for applications (menus, toolbars, dock windows)
- Easy communication between application components (signals and slots)
- Unified painting system (with transparency, anti-aliasing and SVG support)
- Rich text processing, display and printing
- Database support (SQL) and model/view features
- Input/output and networking
- Other features
  - Container classes
  - Threading
  - Resources
  - XML processing

PyQt

Qt 3: PyQt is a set of bindings for Qt 3 by Phil Thompson at Riverbank Computing.

- Uses SIP to generate bindings
- Comprehensive API coverage
- Dual licensed under the GPL and Commercial licenses
- Community mailing list with around 500 members
- Wiki at http://www.diotavelli.net/PyQtWiki

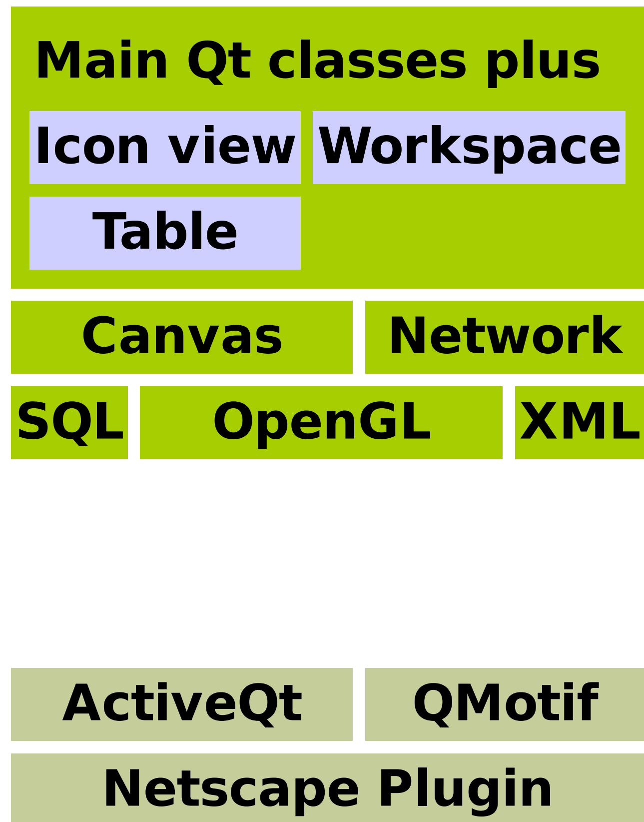
PyQt3	1998: First release 2000: Windows release 2002: Mac OS X and Zaurus releases
PyQt4	2006: PyQt4 release

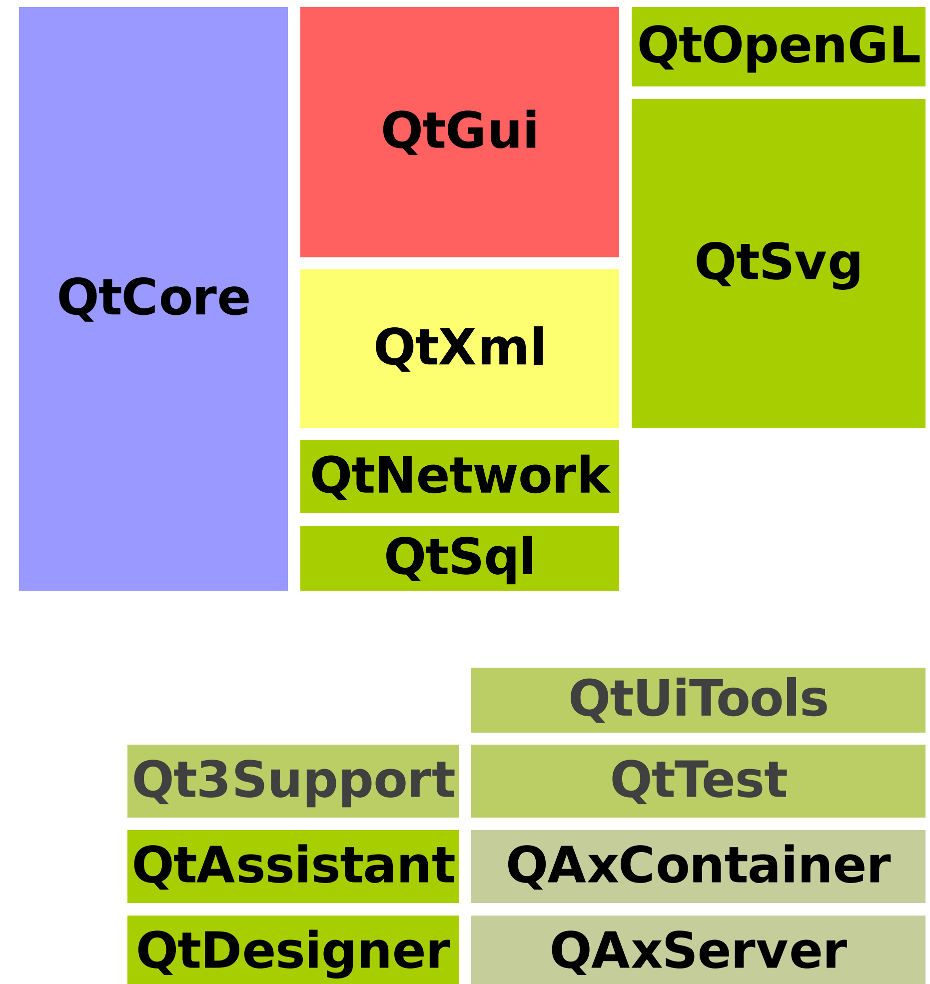
PyKDE

KDE 3: PyKDE is a set of bindings for KDE 3 by Jim Bublitz that supports these mainstream KDE libraries:

- **DCOP** – interprocess communication
- **kdecore** – application and configuration
- **kdeui** – widgets, dialogs, user interface elements
- **khtml** – HTML display (used by Konqueror and Safari)
- **kio** – network transparent communications
- **kparts** – high-level reusable GUI components
- **kdeprint** – printing, dialogs, print jobs and management
- Others (**kfile**, **kmdi**, **kspell**, **kdesu**, **kutils**)

Library Structure





QtCore

- Object and meta-object system:
  - **QObject**, **QMetaObject**
- Basic value types:
  - **QByteArray**, **QString**, **QDate**, **QTime**, **QPoint[F]**, **QSize[F]**
- File system abstraction:
  - **QFile**, **QDir**, **QIODevice**, **QTextStream**, **QDataStream**
- Basic application support:
  - **QCoreApplication** – encapsulates an application
  - **QEvent** – communication (see also **signals and slots**)
  - **QTimer** – signal-based timed event handling

QtGui

- Widgets:
  - **QCheckBox**, **QComboBox**, **QDateTimeEdit**, **QLineEdit**, **QPushButton**, **QRadioButton**, **QSlider**, **QSpinBox**, etc.
- Basic value types:
  - **QColor**, **QFont**, **QBrush**, **QPen**
- Painting system and devices:
  - **QPainter**, **QPaintDevice**, **QPrinter**, **QImage**, **QPixmap**, **QWidget**
- Basic application support:
  - **QApplication** – encapsulates a GUI application
- Rich text:
  - **QTextEdit**, **QTextDocument**, **QTextCursor**

QtGui

- Display widgets
- Input widgets
- Text entry widgets
- Buttons
- Scrolling list and tree widgets
- Tab widgets

Meeting
Date: Jul 3, 2006
Time: 9:00:00 AM
Location: Main Conference Room
Participants: Alice
Andrew
Bob
Carol
David
Edward
Felicity
George
Henry
Isabel
Jeremy
Katherine

QGroupBox
QLabel
QDateEdit
QTimeEdit
QComboBox
QListWidget

Options
Find text
Find images
Interactive search

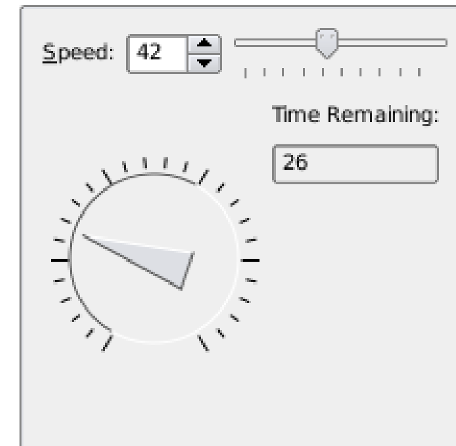
QGroupBox
QRadioButton
QCheckBox

Contact | Presentation
Name: David Boddie
Address: Trolltech ASA
Sandakerveien 116
PO Box 4332 Nydalen
NO-0402 Oslo
Norway

QTabWidget
QLabel
QLineEdit
QTextEdit

QtGui

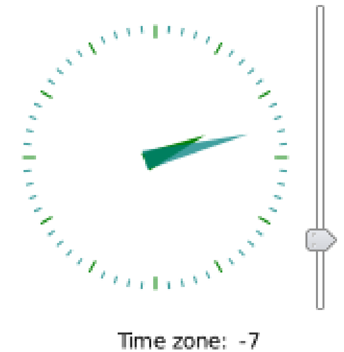
- Range controls
- Tables
- Scrolling views
- Database support
- Custom widgets



QFrame
QLabel
QSpinBox
QSlider
QDial

	Year	Location
1	2002	Charleroi, Belgium
2	2003	Charleroi, Belgium
3	2004	Gothenburg, Sweden
4	2005	Gothenburg, Sweden
5	2006	CERN, Geneva, Switzerland

QTableWidget



AnalogClock
QSlider
QLabel

Using Widgets

Creating a top-level widget

```
window = QWidget()
window.resize(480, 360)
window.show()
```

- Creates a widget
- Resizes and shows it

Creating a child widget

```
button = QPushButton("Press me", window)
button.move(200, 200)
button.show()
```

- Creates a button inside the window
- Positions and shows it

Placing widgets in a layout

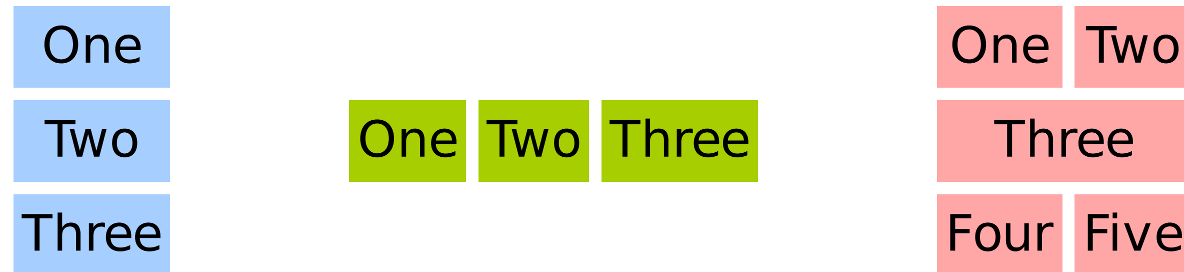
```
okButton = QPushButton("&OK")
cancelButton = QPushButton("&Cancel")

layout = QVBoxLayout()
layout.addWidget(okButton)
layout.addWidget(cancelButton)
window.setLayout(layout)
```

- Creates parent and child widgets
- Creates a layout to arrange widgets
- Adds the child widget to the layout

Using Layouts

Layouts manage child widgets and are responsible for:

- Updating their sizes and positions
- Providing default and minimum sizes



Horizontal, vertical and grid layouts

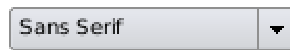
```
yesButton = QPushButton("&Yes")
noButton = QPushButton("&No")

layout = QHBoxLayout()
layout.addStretch(1)
layout.addWidget(yesButton)
layout.addWidget(noButton)
```

```
nameLabel = QLabel("Name:")
nameEdit = QLineEdit()
addressLabel = QLabel("Address:")
addressEdit = QTextEdit()

layout = QGridLayout()
layout.addWidget(nameLabel, 0, 0)
layout.addWidget(nameEdit, 0, 1)
layout.addWidget(addressLabel,0,0,Qt.AlignTop)
layout.addWidget(addressEdit, 0, 1)
```

Signals and Slots

Signals and slots allow objects to communicate with each other via type-safe interfaces.

QComboBox



activated(QString)

setFontFamily(QString)

QTextEdit

- Sender and receiver do not need to know about each other
- Connections can be direct or queued
- Sender and receiver can be in different threads

Signals and Slots

Making connections

```
class MainWindow(QMainWindow):

  def __init__(self):

    QMainWindow.__init__(self)

    fileMenu = self.menuBar().addMenu(self.tr("&File"))
    exitAction = fileMenu.addAction(self.tr("E&xit"))

    helpMenu = self.menuBar().addMenu(self.tr("&Help"))
    aboutAction = helpMenu.addAction(self.tr("&About This Example"))

    self.connect(exitAction, SIGNAL("triggered()"), qApp, SLOT("quit()"))
    self.connect(aboutAction, SIGNAL("triggered()"), self.showAboutBox)

    # Set up the rest of the window.

  def showAboutBox(self):

    QMessageBox.information(self, self.tr("About This Example"),
        self.tr("This example shows how signals and slots are used to\n"
                "communication between objects in Python and C++."))
```

Writing an Application

Creating an application

```
app = QApplication(sys.argv)
window = MainWindow()
window.show()
sys.exit(app.exec_())
```

- Creates the application
- Creates and shows the main window (a **`QMainWindow`** subclass)
- Runs the event loop then exits

Running an application in different styles

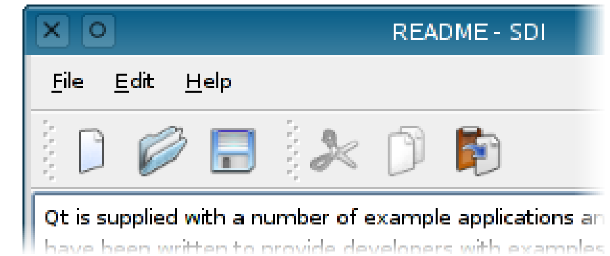
```
myapplication -style plastique
myapplication -style motif
myapplication -style windows
```

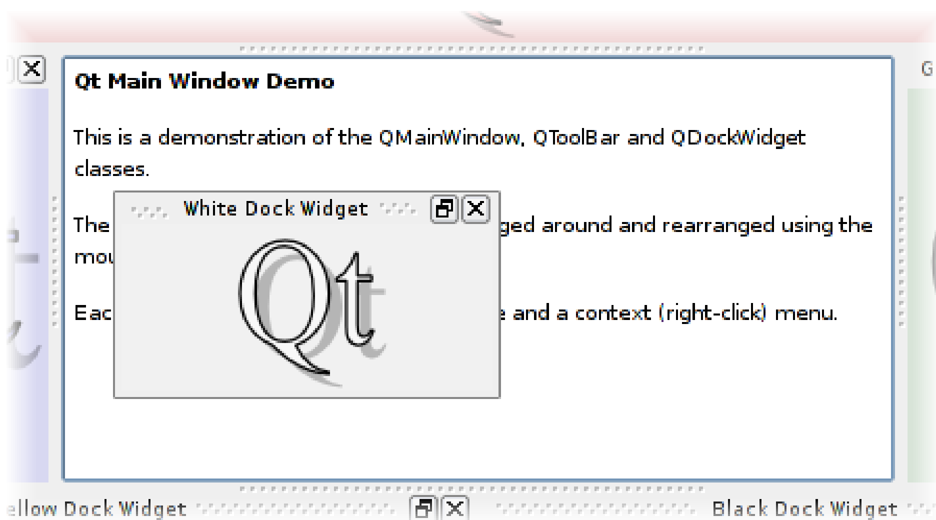
- **`QApplication`** parses the command line arguments
- The `-style` option can be used to override the native style

Main Window Classes

Main windows usually have

- Menus – built using **QMenu** and populated with actions
- Toolbars – built using **QToolBar**, these usually share actions with menus





- Dock windows – provided by **QDockWidget**
- A status bar – provided by **QStatusBar**
- A central widget containing the main GUI

- User actions are represented by the **QAction** class
- The action system synchronizes menus, toolbars, and keyboard shortcuts
- It also stores information about tooltips and interactive help

Actions

To create an action, you can:

- Instantiate a **QAction** object directly
- Call **addAction()** on existing **QMenu** and **QToolBar** objects

Then you can share it with other objects.

Sharing actions

```
self.saveAction = QAction(QIcon(":/images/save.png"), self.tr("&Save..."), self)
self.saveAction.setShortcut(self.tr("Ctrl+S"))
self.saveAction.setStatusTip(self.tr("Save the current form letter"))
self.connect(self.saveAct, QtCore.SIGNAL("triggered()"), self.save)
...
self.fileMenu = self.menuBar().addMenu(self.tr("&File"))
self.fileMenu.addAction(self.saveAction)
...
self.fileToolBar = self.addToolBar(self.tr("File"))
self.fileToolBar.addAction(self.saveAct)
```

Multiple Document Interface

Applications are designed with different user interfaces:

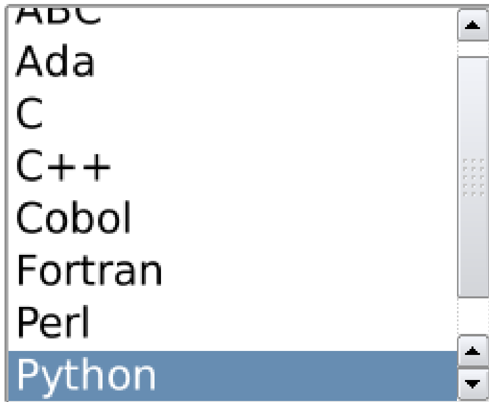
- Single Document Interface (SDI) applications use multiple main windows, each containing a suitable central widget
- Multiple Document Interface (MDI) applications use a **QWorkspace** as the central widget

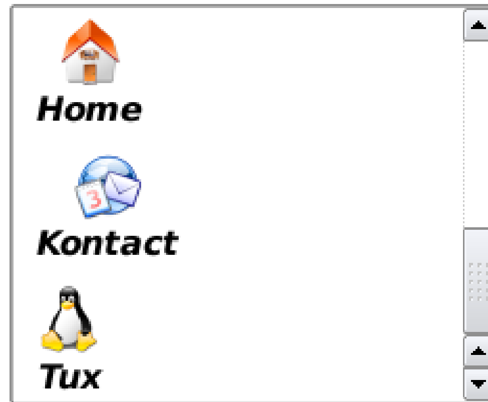
Multiple Document Interface

```
workspace = QWorkspace()
workspace.setWindowTitle("Simple Workspace Example")

for i in range(5):

    textEdit = QTextEdit()
    textEdit.setPlainText("PyQt4 "*100)
    textEdit.setWindowTitle("Document %i" % i)
    workspace.addWindow(textEdit)

workspace.cascade()
```

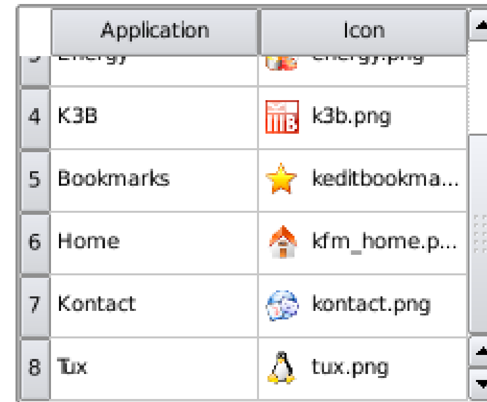
Item Views

Item views are complex controls that handle collections of items, each representing a piece of data.

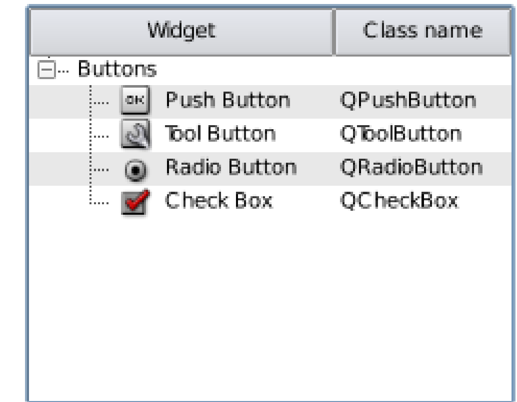




QListView or **QListWidget**



QTableView or **QTableWidget**



QTreeView or **QTreeWidget**

- Qt 3's item views are populated with item objects
- Qt 4 has item-based view classes **and** a model/view framework

Item Views

What's the difference between the item-based and model-based approaches?

Item-based tree of items

```
tree = QTreeWidget()
tree.setColumnCount(2)
tree.setHeaderLabels(["Name", "Address"])

for name, address in phonebook.items():
    item = QTreeWidgetItem(tree)
    item.setText(0, name)
    item.setText(1, address)

tree.show()
```

- Items are easy to use
- You just create them and add them to parent widgets or items
- It all has to be done by you

Model-based version

```
# Given an existing model...

tree = QTreeView()
tree.setModel(model)
tree.show()

table = QTableView()
table.setModel(model)
table.show()
```

- Models automatically populate views with items
- Views (of different kinds) can share models
- We left out the tricky part...

Models and Views

Concepts

- Models hold data for views to display
- Views access data using **indexes**
- Delegates display individual items for views
- Roles describe the types of data

```
# Reading
index = model.index(row, column, parent)
data = index.data(index, role)

# Writing
model.setData(index, data, role)
```

- Models expose pieces of data as items in tables
- Items can expose tables of child items

Models and Views

A simple model

```
class ImageModel(QAbstractTableModel):

  def __init__(self, image, parent=None):

      QAbstractTableModel.__init__(self, parent)
      self.image = QImage(image)

  def rowCount(self, parent):

      return self.image.height()

  def columnCount(self, parent):

      return self.image.width()

  def data(self, index, role):

      if not index.isValid():
        return QVariant()
      elif role != QtCore.Qt.DisplayRole:
        return QVariant()

      return QVariant(qGray(
          self.image.pixel(index.column(), index.row())))
```

Models and Views

With a suitable model, views can be used to display any kind of data:

- XML data can be displayed in a tree view
- There's already a Qt example of a DOM-based XML model
- Torsten decided to write an ElementTree model

Torsten's ElementTree model

- Around 50 lines of code
- Read-only
- Copes with quite large files
- Fast, even compared to pure C++ models

Database Support

- Like Python, Qt 4 has classes for working with databases
- These are integrated with the model/view framework

Accessing a database with a SQL table model

```
db = QSqlDatabase.addDatabase("QSQLITE")
db.setDatabaseName(databaseName)

model = QSqlTableModel(self)
model.setTable("person")

model.setEditStrategy(QSqlTableModel.OnManualSubmit)
model.select()

view1 = QTableView(self)
view1.setModel(model)
view2 = QTableView(self)
view2.setModel(model)
```

Database Support

Python has its own standard database API:

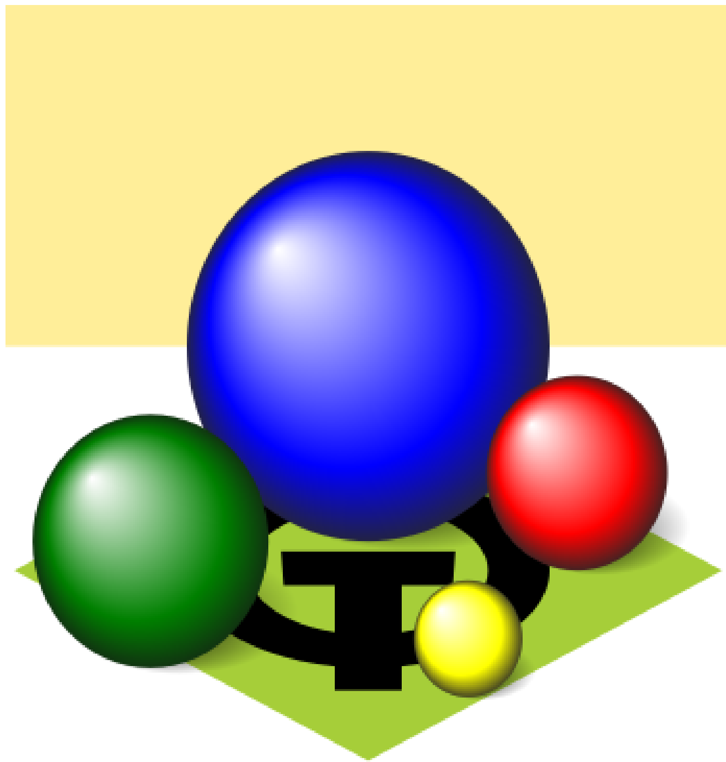
- Included with many Python database modules
- Many Python developers are familiar with it
- Torsten decided to write a model for that, too

Accessing a database with a SQL table model

Scalable Vector Graphics (SVG)

SVG support can be accessed in two ways:

- You can use **QSvgWidget** to load and display pictures in a widget
- You can use **QSvgRenderer** to load and render pictures to *any* paint device
- SVGs can also contain animations



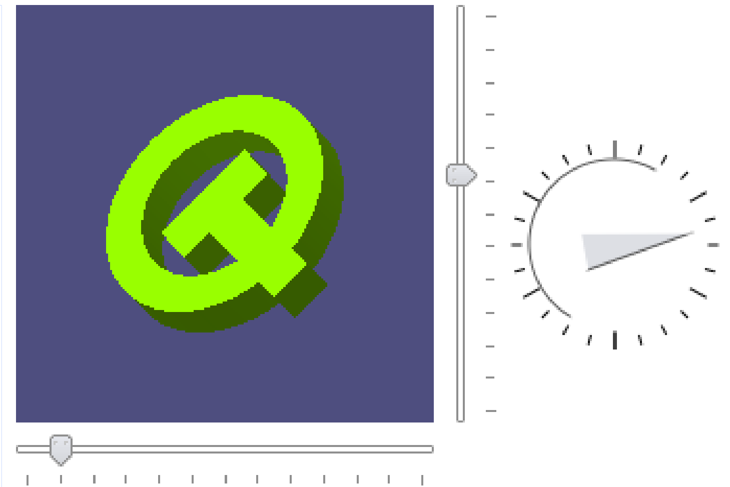
```
# Showing an SVG drawing:
widget = QSvgWidget(parent)
widget.load(filename)
widget.show()

# Rendering a drawing on an image:
pixmap = QPixmap(200, 200)
renderer = QSvgRenderer()
painter = QPainter()
painter.begin(pixmap)
renderer.render(painter)
painter.end()
```

OpenGL Integration

Qt provides a **QGLWidget** (a **QWidget** subclass) to display OpenGL content:

- The OpenGL context is handled automatically
- Convenience functions to handle textures and colors

```
class GLWidget(QGLWidget):
    def __init__(self, parent):
        QGLWidget.__init__(self, parent)
    def initializeGL(self):
        # Set up display lists, OpenGL options.
    def paintGL(self):
        # Clear buffers, apply transformations,
        # paint.
    def resizeGL(self):
        # Resize viewport, recalculate matrices.
```



Custom widgets

Widgets can be combined to make composite widgets by subclassing an existing widget class.

```
class AddressWidget(QWidget):

  def __init__(self, parent = None):

    QWidget.__init__(self, parent)

    nameLabel = QLabel(self.tr("Name:"))
    nameEdit = QLineEdit()
    addressLabel = QLabel(self.tr("Address:"))
    addressEdit = QTextEdit()
```

In the `__init__` method:

- Call the base class's `__init__` method
- Create child widgets

Use layouts to make the contents resize nicely.

```
    layout = QGridLayout()
    layout.addWidget(nameLabel, 0, 0)
    layout.addWidget(nameEdit, 0, 1)
    layout.addWidget(addressLabel, 1, 0, Qt.AlignTop)
    layout.addWidget(addressEdit, 1, 1)

    self.setLayout(layout)
```

- Lay out the child widgets

Custom widgets

Custom widgets can also be built from scratch.

- To make new controls
- For decorative purposes

```
class CustomWidget(QWidget):

  def __init__(self, parent = None):

    QWidget.__init__(self, parent)
    # Initialize the widget.

  def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    # Do some painting.
    painter.end()

  def sizeHint(self):

    return QSize(200, 200)
```

In the `__init__()` method:

- Subclass an existing widget class
- Call the base class's `__init__()` method

In the `paintEvent()` method:

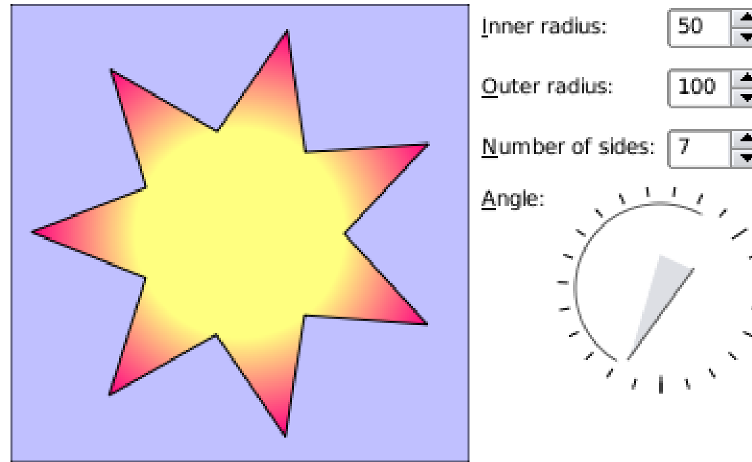
- Widgets are paint devices
- Just use a painter to draw on them

In the `sizeHint()` method:

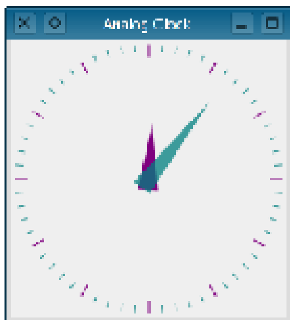
- Return a preferred size to help the layout engine

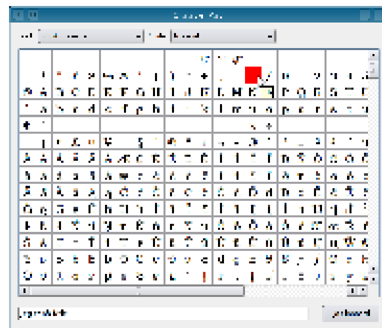
Custom widgets

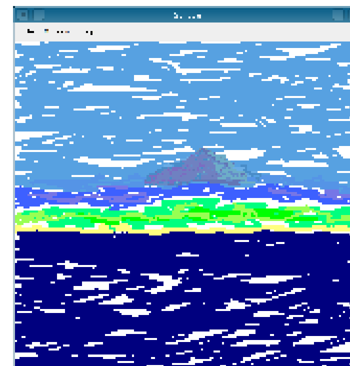
They can also provide their own signals and slots...



There are lots of examples of custom widgets:



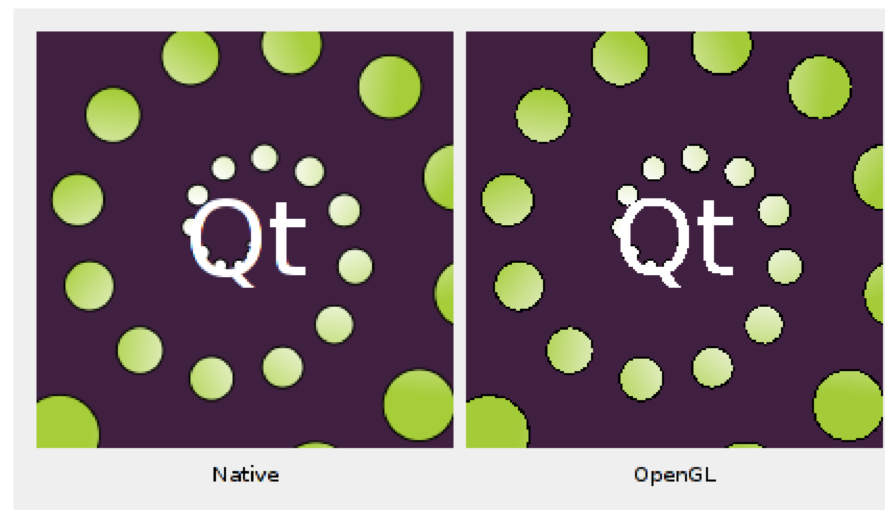




OpenGL Integration Revisited

You can also use **`QPainter`** to paint onto a **`QGLWidget`**:

- Painting operations are translated to OpenGL calls
- The result is accelerated 2D rendering



- Relies on extensions for anti-aliasing, so you may be trading looks for speed

Qt Designer

Qt Designer is Trolltech's design tool for creating user interfaces.

Forms created with Qt Designer are stored in XML (.ui) files

- These can be compiled to C++ with *uic*
- You can also use *pyuic4* to convert them to Python
- Or you can use the Python **`uic`** module to generate the GUI at run-time

This presentation was created with Qt Designer.

The GUI is shown using PyQt4.

Thanks

- Trolltech for Qt
- Phil Thompson for SIP and PyQt4
- Jim Bublitz for PyKDE
- Paul Boddie for suggestions
- The PyQt community (especially those on the mailing list)

Links

Trolltech: http://www.trolltech.com

Riverbank Computing: http://www.riverbankcomputing.com/

PyQt Wiki at http://www.diotavelli.net/PyQtWiki