EUROPYTHON 2011
FLORENCE, JUNE 20–26

Developing cutting-edge applications
With PyQt

develer

. software . hardware . innovation

What is Qt?

Everything you need to create web-enabled desktop, mobile and embedded applications.

Qt is a cross-platform application and UI Framework.

Qt Modules

QtCore QtNetwork QtGui QtScript QtDeclarative

QtTest QtXml QtOpenGL QtWebKit

QtXmlPatterns QtSvg

QtSql QtMultimedia

What is PyQt?

Everything you need to create web-enabled desktop applications.

PyQt is a set of Python bindings for Nokia's Qt application framework and runs on all platforms supported by Qt.

SIP is a tool that makes it very easy
To create Python bindings
For C and C++ libraries.

PyQt v4 is available on all platforms
Under GNU GPL (v2 and v3) and a commercial lincese.
Unlink Qt, PyQt v4 is not available under the LGPL.

Get Ready!

(How to Install Qt4 & PyQt4)

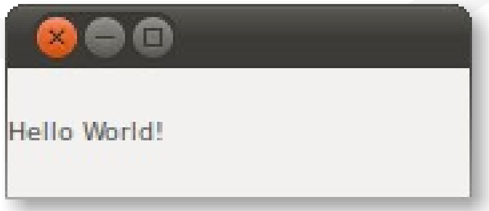
http://www.riverbankcomputing.co.uk/software/pyqt/download

```
apt-get install python-qt4
yum install PyQt4
emerge dev-python/PyQt4
```

PyQt Hello World

hello/hello_world.py

```
from PyQt4.Qt import *

if __name__ == "__main__":
    app = QApplication([])

    label = QLabel("Hello World!")
    label.show()

    app.exec_()
```


Hello World!

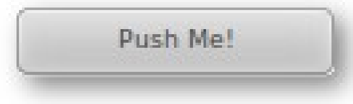
What is a Widget?

- User interface object that represents an action and/or displays a piece of information
- Its value can be changed both programmatically (by the application) and by user-driven interaction.

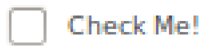
QWidget

- Base class for all widgets
- Receives events from the outside windowing system and draw itself
- Communication with the outside world occur via notifications (signals) and available actions (slots)

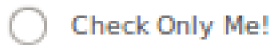
Enter in a World of Widgets



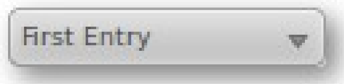
QPushButton("Push Me!")



QCheckButton("Check Me!")



QRadioButton("Check Only Me!")

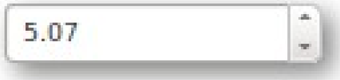


```
combo = QComboBox()
combo.addItem("First entry")
combo.addItem("Second entry");
```

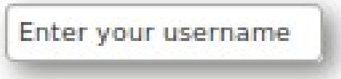
develer

. software . hardware . innovation

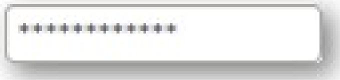
Enter in a World of Widgets



```
spin = QDoubleSpinBox()
spin.setValue(5.07)
```



```
line_edit = QLineEdit()
line_edit.setText(“Enter your username”)
...
username = line_edit.text()
```



```
line_edit = QLineEdit()
line_edit.setEchoMode(QLineEdit.Password)
...
password = line_edit.text()
```

```
text_edit = QTextEdit()
text_edit.setHtml(“<h1>My Text</h1> Prova”);
text_edit.setPlainText(“Hello”);
```

develer

. software . hardware . innovation

How can I interact with these widgets?

Signals & Slots

Each object exposes
a set of signals (notifications)
and a set of slots (actions)

```
from PyQt4.Qt import *

def _onClick():
    print "Button Clicked!"

if __name__ == "__main__":
    app = QApplication([])

    button = QPushButton("Push Me!")
    QObject.connect(button, SIGNAL("clicked()"), _onClick)
    button.show()

    app.exec_()
```

Signals & Slots

Each object exposes
a set of signals (notifications)
and a set of slots (actions)

```
def _onClick():
    print "Button Clicked!"
```

Sender

Signal

```
QObject.connect(button, SIGNAL("clicked()"), _onClick)
button.show()

app.exec_()
```

Slot
(Signal Callback)

Signals & Slots

- Each object exposes a set of signals (notifications) and a set of slots (actions).
- Externally, signals can be connected to slots
- A signal is “emitted” when an object changes its internal state in a way that might be interesting to others
- A slot is an action, implemented by member function which might be connected to a signal (or used directly!)

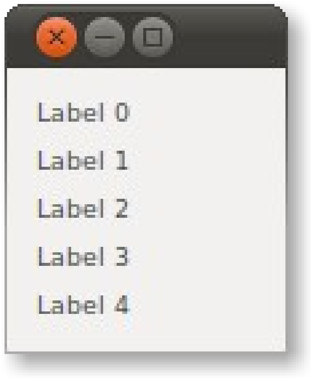
Group Widgets toghether!

(How to build a real UI)

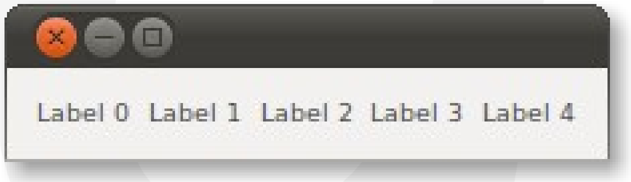
Positions of controls

- How do you position controls within a form?
    - generically: children within their parent
- Old-skool solution: absolute positions x, y
  - Impossibile to write GUI code by hand
  - Impossible for users to stretch dialogs
- Qt supports absolute positions (.move(), .resize()) but gives a far batter solution. Automatic Position!

Horizontal/Vertical Grouping!



```
def buildLayout():
   vlayout = QVBoxLayout()
   for i in range(5):
      vlayout.addWidget(QLabel("Label %d" % i))
   return vlayout
```



```
def buildLayout():
   hlayout = QHBoxLayout()
   for i in range(5):
      hlayout.addWidget(QLabel("Label %d" % i))
   return hlayout
```

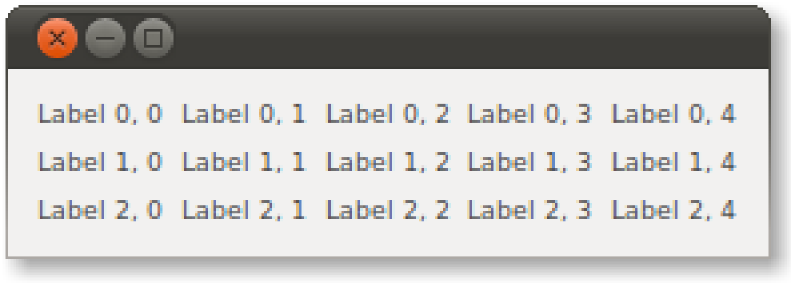
```
if __name__ == "__main__":
   app = QApplication([])

   w = QWidget()
   w.setLayout(buildLayout())
   w.show()

   app.exec_()
```

QBoxLayout takes the space available divides it up into a row of boxes, and makes each managed widget fill one box.

Grid grouping!

The QGridLayout class lays out widgets in a grid.
It takes the space available, divides it up into rows and columns,
and puts each widget it mnages into the correct cell.

```
def buildLayout():
   grid_layout = QGridLayout()
   for row in range(3):
       for col in range(5):
           label = QLabel("Label %d,%d" % (row, col))
         grid_layout.addWidget(row, col, label)
  return grid_layout
```



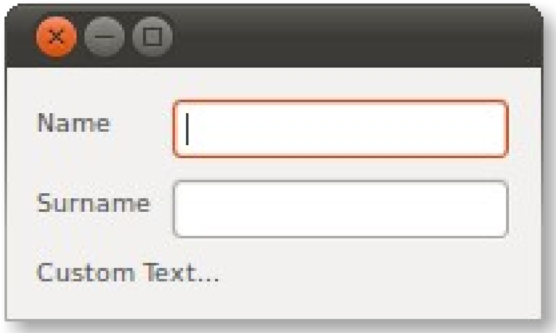
```
Label 0, 0  Label 0, 1  Label 0, 2  Label 0, 3  Label 0, 4
Label 1, 0  Label 1, 1  Label 1, 2  Label 1, 3  Label 1, 4
Label 2, 0  Label 2, 1  Label 2, 2  Label 2, 3  Label 2, 4
```

```
if __name__ == "__main__":
   app = QApplication([])

   w = QWidget()
   w.setLayout(buildLayout())
   w.show()

   app.exec_()
```

Form Grouping!

QFormLayout lays out its children in a two-column form.
The left column consists of labels
and the right column consists of "field" widgets
(line editors, spin boxes, etc.)

```
def buildLayout():
  form_layout = QFormLayout()
  form_layout.addRow("Name", QLineEdit())
  form_layout.addRow("Surname", QLineEdit())
  form_layout.addRow(QLabel("Custom text..."))
  return form_layout
```



Name

Surname

Custom Text...

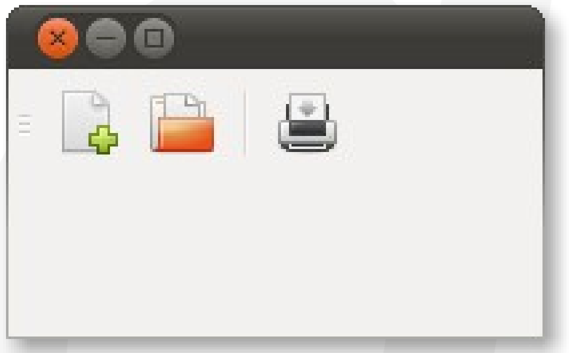
```
if __name__ == "__main__":
    app = QApplication([])

    w = QWidget()
    w.setLayout(buildLayout())
    w.show()

    app.exec_()
```

Main Window & Dialogs

Menubar, Toolbar, and ...

Toolbar

widget/toolbar.py

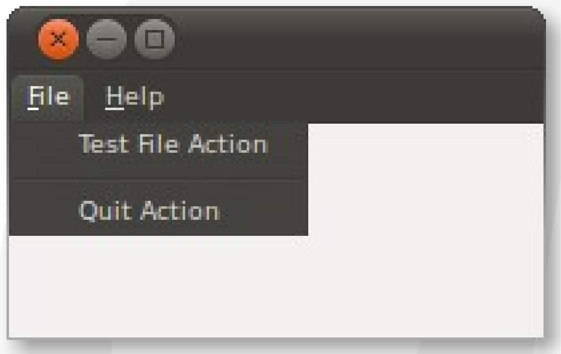


```
def onAction(n):
    print 'Clicked Action', n

if __name__ == '__main__':
    app = QApplication([])

    main_window = QMainWindow()

    tool_bar = main_window.addToolBar('MainToolbar')
    action1 = tool_bar.addAction(QIcon.fromTheme('document-new'), 'Action 1')
    action2 = tool_bar.addAction(QIcon.fromTheme('document-open'), 'Action 2')
    tool_bar.addSeparator()
    action3 = tool_bar.addAction(QIcon.fromTheme('document-print'), 'Action 3')

    QObject.connect(action1, SIGNAL('triggered()'), lambda: onAction(1))
    QObject.connect(action2, SIGNAL('triggered()'), lambda: onAction(2))
    QObject.connect(action3, SIGNAL('triggered()'), lambda: onAction(3))

    main_window.show()

    app.exec_()
```

Menubar

widget/menubar.py

```
def fileAction():
    print 'Clicked File Action'

def helpAction():
    print 'Clicked Help Action'

if __name__ == '__main__':
    app = QApplication([])

    main_window = QMainWindow()

    menu_bar = main_window.menuBar()
    file_menu = menu_bar.addMenu("&File")
    file_action = file_menu.addAction('Test File Action')
    file_menu.addSeparator()
    quit_action = file_menu.addAction('Quit Action')

    help_menu = menu_bar.addMenu("&Help")
    help_action = help_menu.addAction('Test Help Action')

    QObject.connect(file_action, SIGNAL('triggered()'), fileAction)
    QObject.connect(quit_action, SIGNAL('triggered()'), app.quit)
    QObject.connect(help_action, SIGNAL('triggered()'), helpAction)

    main_window.show()

    app.exec_()
```



Dialogs

- Dialogs are windows that carry out short tasks (e.g. config panels or notifications to the user)
- QDialog
  - They always are top-level widget
  - Will open on center of its parent widget
  - Have a “result” value
  - Two very different kind of dialogs:
    - Modal
    - Modeless

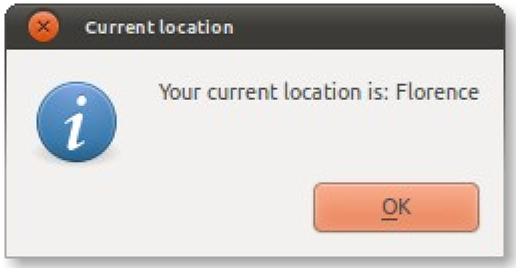
Modal dialogs

- Modal dialogs block input to other windows until the user closes the dialog
  - Runs its own event loop (not concidentally, a modal dialog is started with dialog.exec())
  - QDialog::exec()
    - Accepted/Rejected
    - (slots) accept(), reject()
    - QDialog event loop ends with accept()/reject()
  - Modal dialog explicitly require user intervention

Modeless dialogs

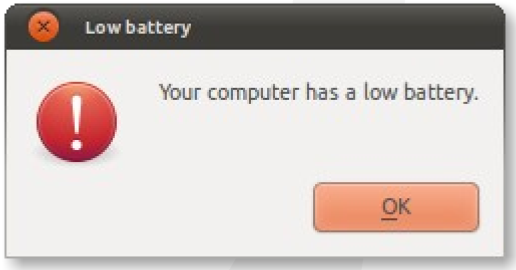
- Operates independently of other windows
  - No internal event loop
  - A modeless dialog is started with dialog.show();
  - Useful for tool windows (think search&replace dialog in word processors)

Default Message Dialogs



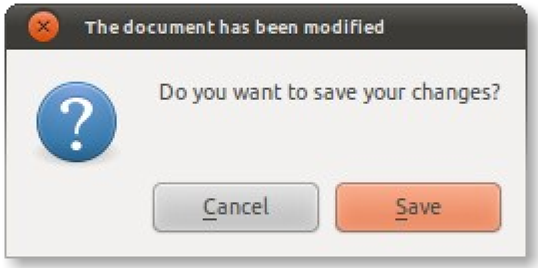
Your current location is: Florence

```
QMessageBox.information(parentWindow,
                        "Current location",
                        "Your current location is: Florence")
```



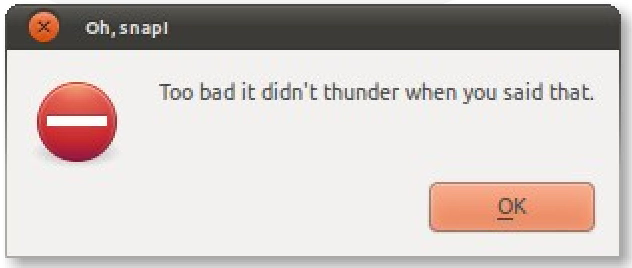
Your computer has a low battery.

```
QMessageBox.warning(parentWindow,
                    "Low battery",
                     "Your computer has a low battery.")
```



Do you want to save your changes?

```
QMessageBox.question(parentWindow,
                     "The document has been modified",
                     "Do you want to save your changes?",
                     QMessageBox.Save | QMessageBox.Cancel)
```



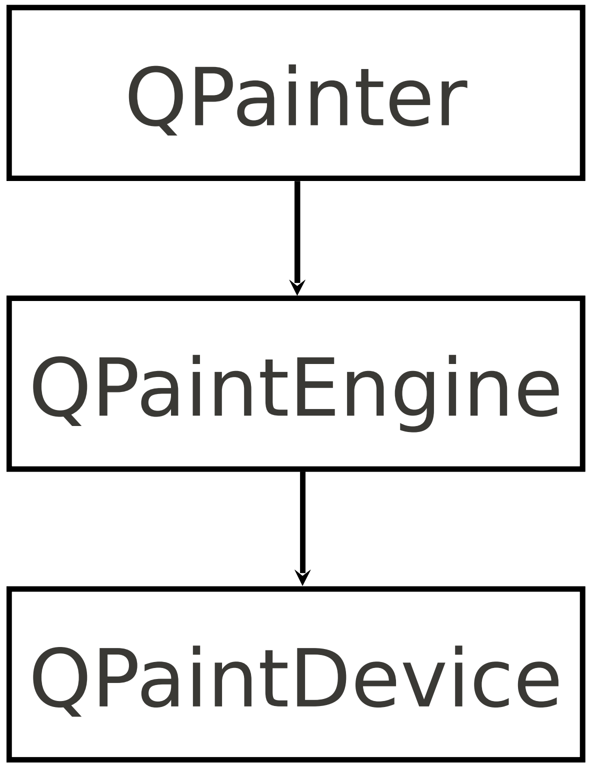
Too bad it didn't thunder when you said that.

```
QMessageBox.critical(parentWindow,
                     "Oh, snap!",
                     "Too bad it didn't thunder when you said that.")
```

The Paint System

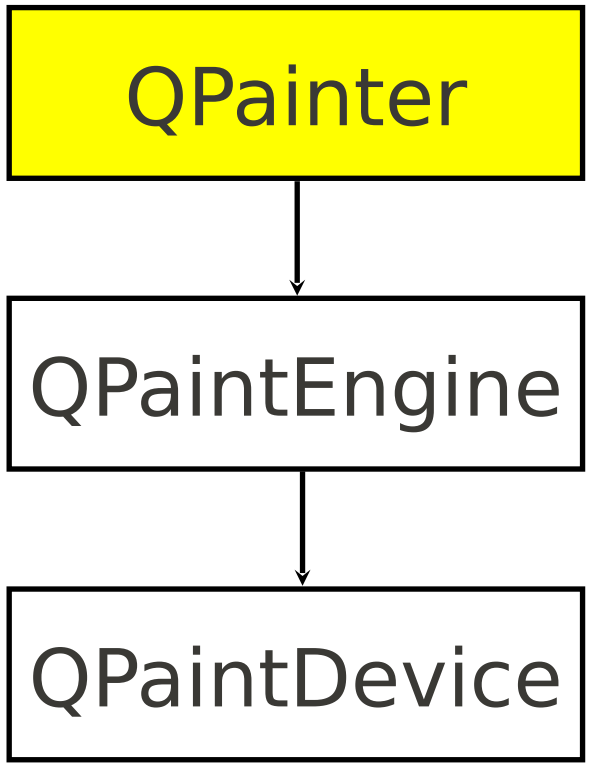
Qt's paint system enables painting
on screen and print devices
Using the same API

Qt Painting System



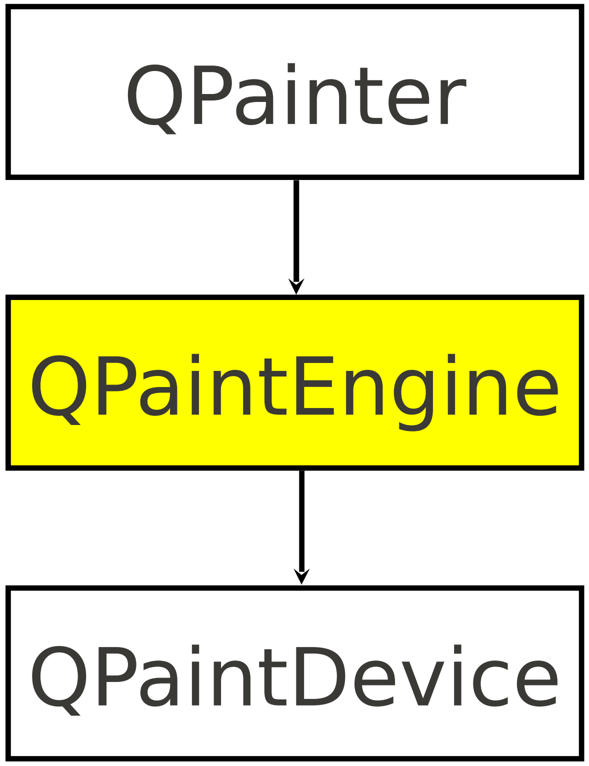
- All kinds of drawing in Qt follow this pipeline
- Both built-in Qt widgets and custom ones.
- Also true for GL contexts, but they can be drawn on using direct GL commands too.

Qt Painting System



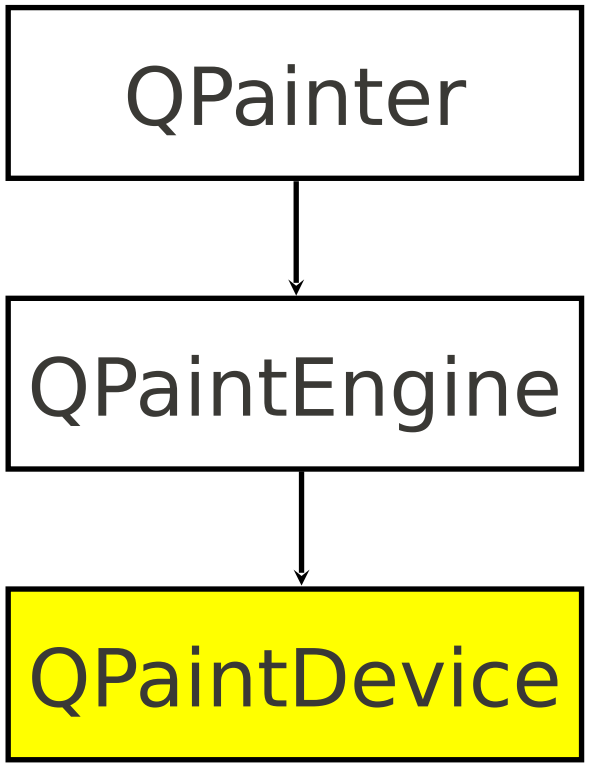
- Implements the drawing of all shapes supported by Qt 2D engine, using the API of QPaintEngine
  - Text, images, geometric primitives, Bézier curves, pie segments...
  - Antialiasing, alpha blending, gradient filling, vector paths...they can be drawn on using direct GL commands too.

Qt Painting System



- Provides a uniform drawing interface
- Draws primitives on painter backends
- Ellipses, lines, points, images, polygons...
- Software emulation for missing features
- Hidden from programmer

Qt Painting System

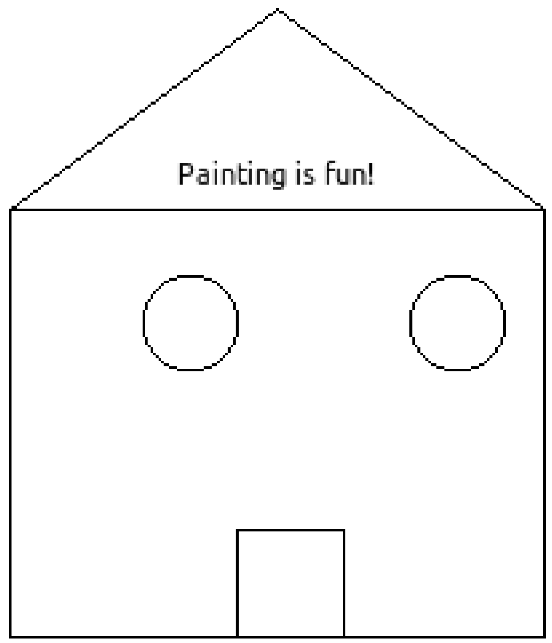


- Base class of all drawable object types (e.g. QWidget is a paint device)
- width, height, dpi, color depth...
- QWidget, QImage, QPixmap, QPrinter, ...

Unleash the Artist in you!

painter/hello_paint.py

```
if __name__ == '__main__':
    app = QApplication([])

    image = QImage(400, 300, QImage.Format_ARGB32)

    painter = QPainter(image)
    painter.fillRect(0, 0, 400, 300, Qt.white)
    painter.drawRect(100, 100, 200, 160)
    painter.drawLine(100, 100, 200, 25)
    painter.drawLine(300, 100, 200, 25)
    painter.drawRect(185, 220, 40, 40)
    painter.drawEllipse(150, 125, 35, 35)
    painter.drawEllipse(250, 125, 35, 35)
    painter.drawText(110, 75, 180, 25,
                     Qt.AlignCenter,
                     "Painting is fun!")
    painter.end()

    image.save('test.png')
```



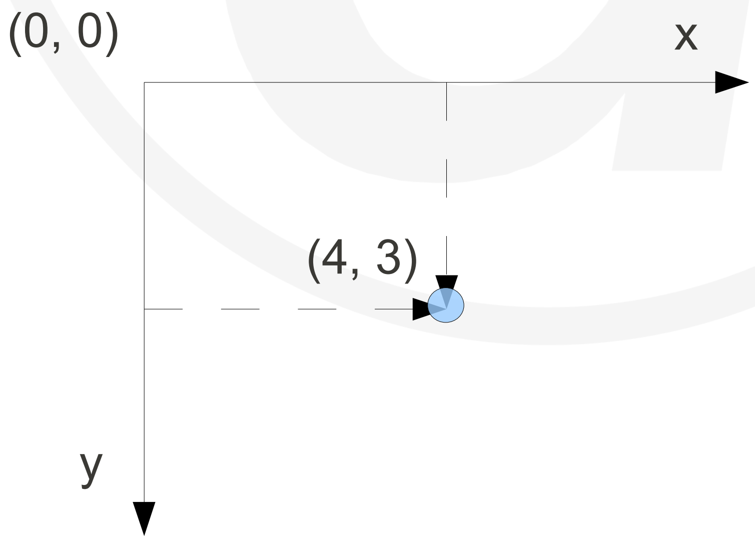
From screen to Pdf, ps, Paper!

painter/printer.py

```
if __name__ == '__main__':
    app = QApplication([])

    printer = QPrinter(QPrinter.HighResolution)
    printer.setOutputFileName('test.pdf')
    printer.setPaperSize(QPrinter.A4)
    printer.setOrientation(QPrinter.Landscape)

    painter = QPainter(printer)

    rect = QRect(100, 100, printer.width() - 200, 200)
    painter.fillRect(rect, Qt.red)
    painter.drawText(rect, Qt.AlignCenter, "Draw on QPainter!")

    painter.end()
```

Coordinate System

- Default coordinate system for QPaintDevices
  - Origin on upper-left corner
  - x values increase to the right, y values increase downwards
- Default unit
  - 1 pixel (raster)
  - 1 point (1/72”) (printers)

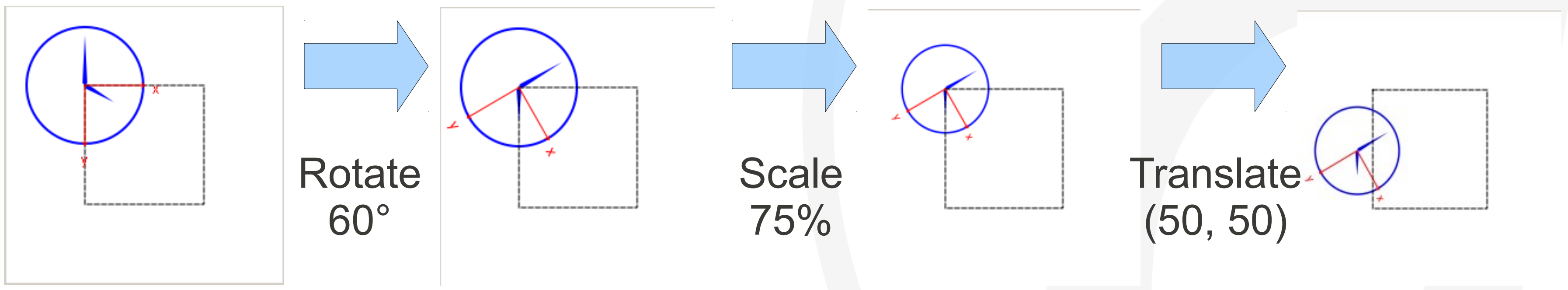


Coordinate Mapping

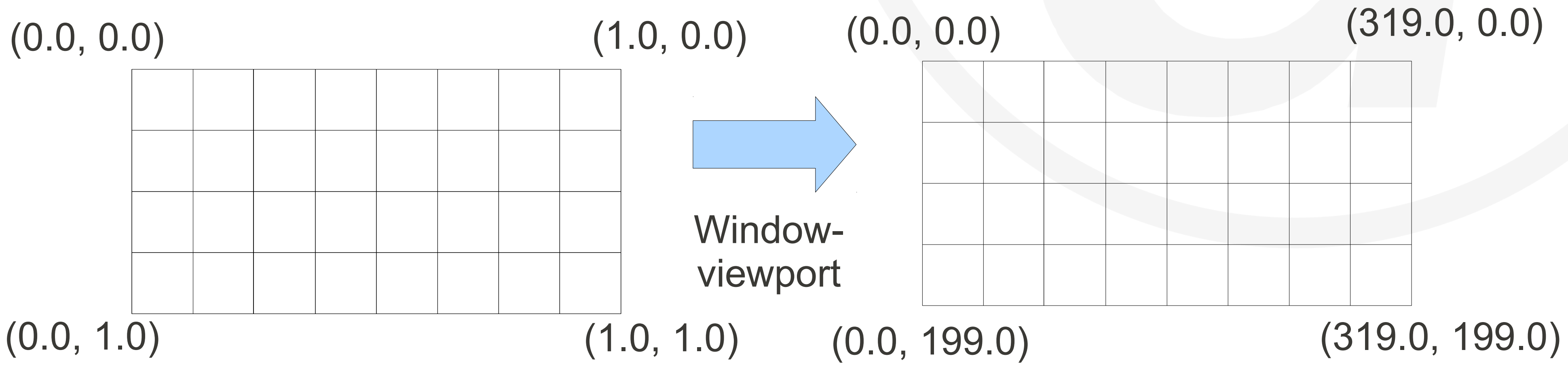
- QPainter works on logical coordinates
- QPaintDevices uses physical coordinates
- By default, they coincide
  - In this case too, Qt does some work under the hood
- Coordinate mapping can be customized using
  - QPainter transformations
  - Window → viewport conversion

Coordinate Mapping

Transformation

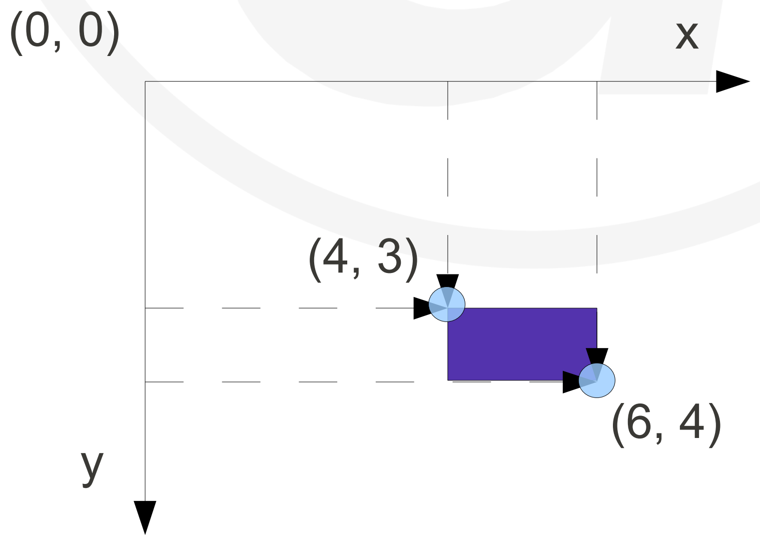


Window-viewport conversion



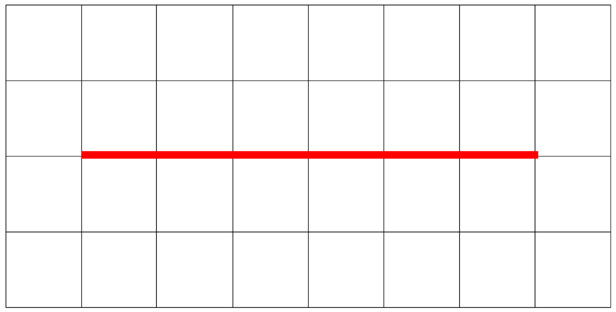
Logical Representation

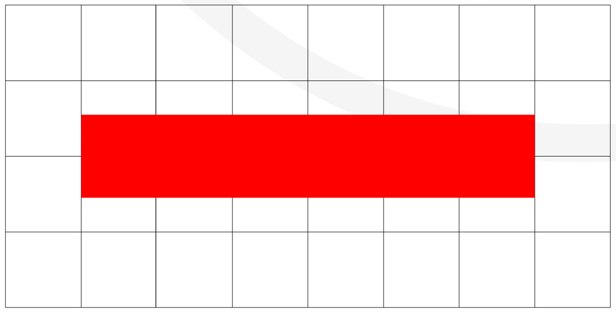
- A logical primitive follows its mathematical model: its size (width, height) and coordinates are not dependent on the device it will be drawn on.
- Rectangle with top (4, 3) and size (2, 1):
  - `QRectF(x, y, width, height);`
  - `QRectF(4.0, 3.0, 2.0, 1.0);`



Physical Representation

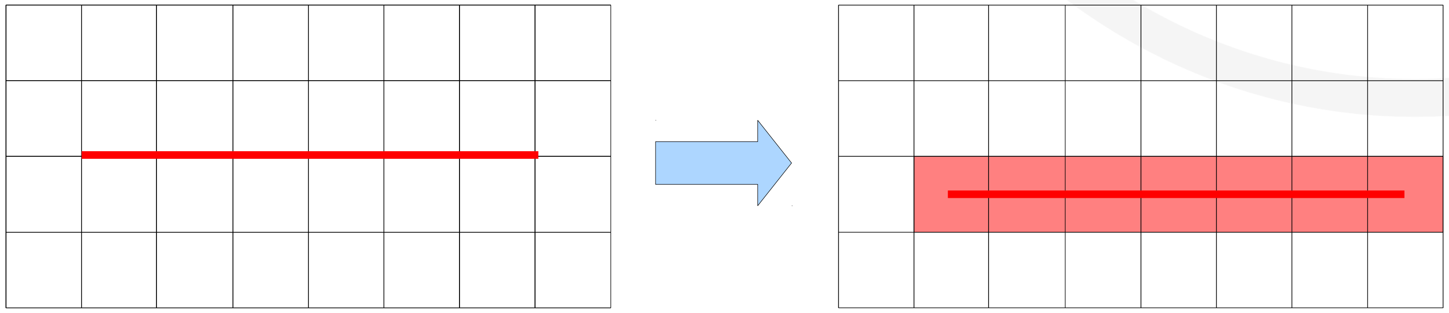
- On real devices, we approximate logical representation using pixel or points
- We are unable to properly represent edges
  - They should lay between two pixel rows
  - Same thing for borders (edges with a size>0)





Physical Representation

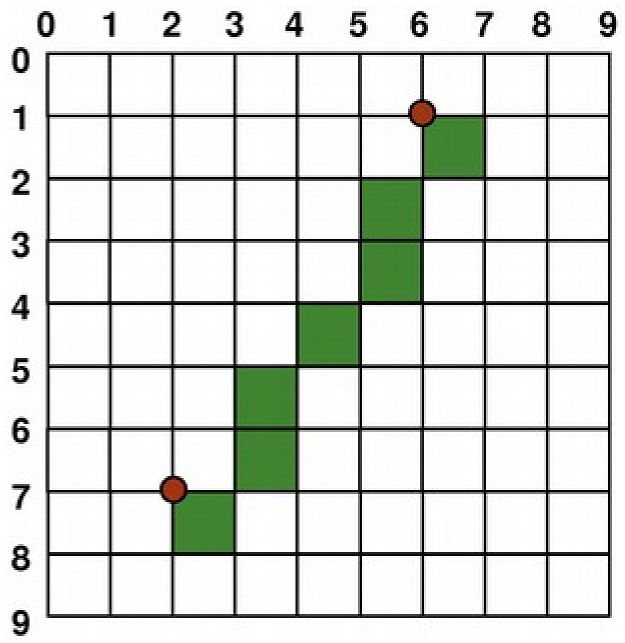
- Qt painters approach the problem depending on rendering mode:
  - Aliased rendering
  - Anti-aliased rendering
- In aliased rendering, physical pixels are drawn using a (+0.5, +0.5) translation on logical ones

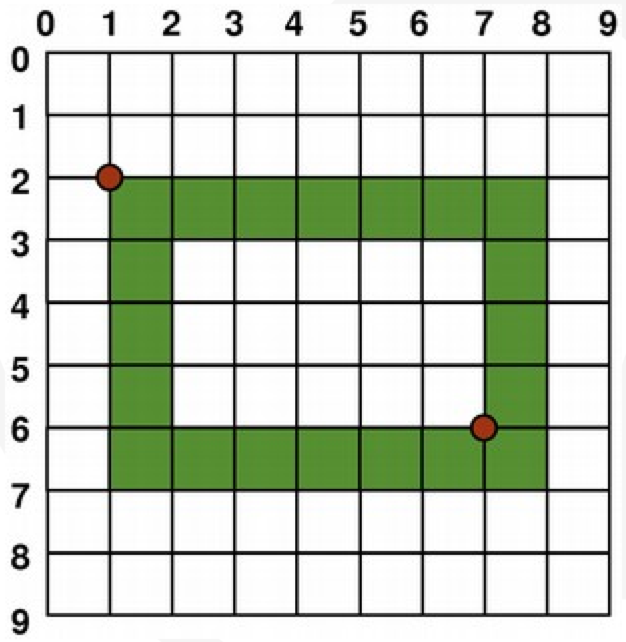


Physical Representation (aliasesd)

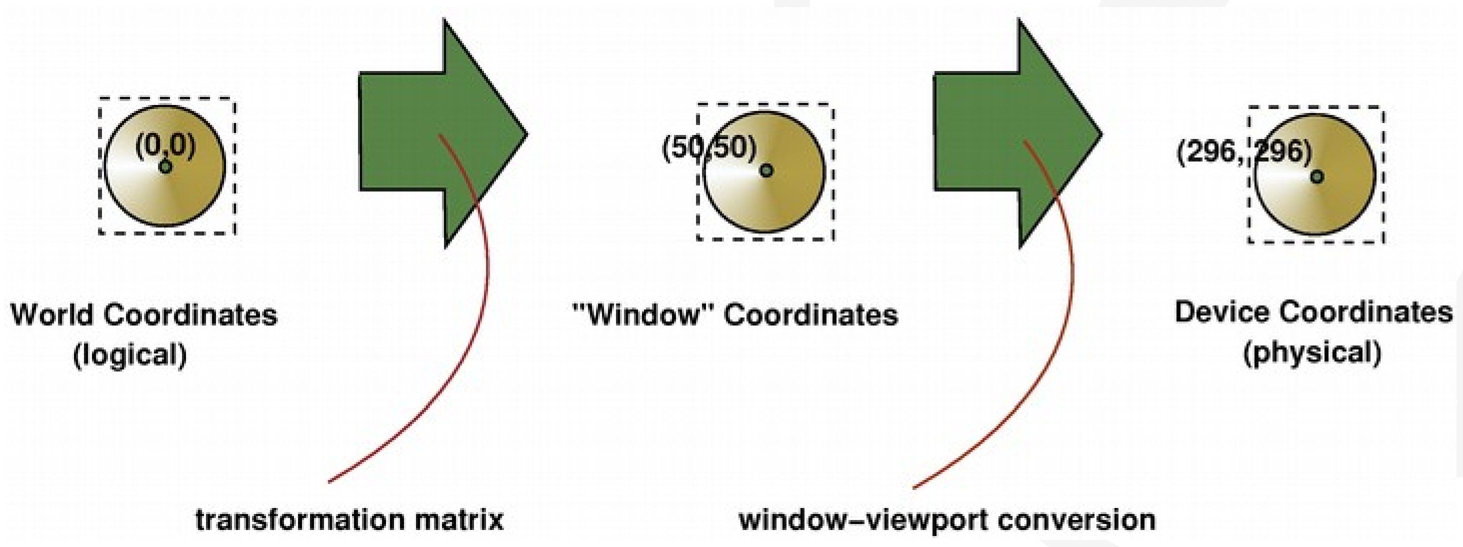
- More generally, aliased rendering follows these rules:
  - Edges: draw +0.5 right below logical pixels (as seen in previous slide)
  - Borders (n pixels wide): draw symmetrically around logical points
  - Borders (n+1 pixels wide): like n pixels width, then render spare pixels +0.5 right below

Physical Representation (aliasesd)





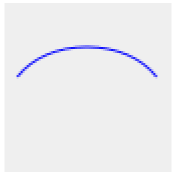
Coordinate transformation



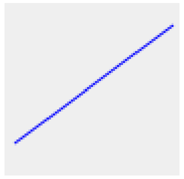
Drawing Features

- QPainter can draw a lot of shapes
- The way they are drawn is influenced by QPainter settings. The most important are:
  - Brush (fills shapes)
  - Pen (draws contours of shapes)
  - Font (draws text)
- All of them are reset when begin() is called

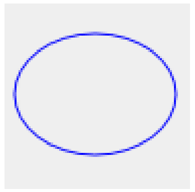
Qpainter Drawing Features



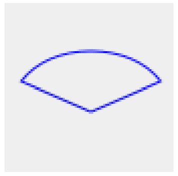
```
Rectangle = QRectF(10.0, 20.0, 80.0, 60.0);
startAngle = 30 * 16;
spanAngle = 120 * 16;
painter = QPainter(self);
painter.drawArc(rectangle, startAngle, spanAngle);
```



```
line = QLineF(10.0, 80.0, 90.0, 20.0);
painter = QPainter(self);
painter.drawLine(line);
```

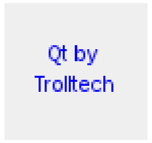


```
rectangle = QRectF(10.0, 20.0, 80.0, 60.0);
painter = QPainter(self);
painter.drawEllipse(rectangle);
```

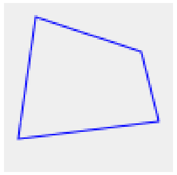


```
rectangle = QRectF(10.0, 20.0, 80.0, 60.0);
startAngle = 30 * 16;
spanAngle = 120 * 16;
painter = QPainter(self);
painter.drawPie(rectangle, startAngle, spanAngle);
```

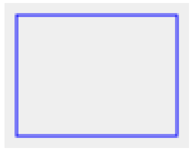
Qpainter Drawing Features



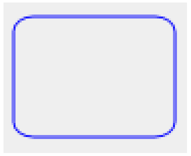
```
painter = QPainter(self);
painter.drawText(rect, Qt.AlignCenter, tr("Qt by\nTrolltech"));
```



```
points = [
    QPointF(10.0, 80.0),
    QPointF(20.0, 10.0),
    QPointF(80.0, 30.0),
    QPointF(90.0, 70.0)
]
painter = QPainter(self);
painter.drawConvexPolygon(points);
```



```
rectangle = QRectF(10.0, 20.0, 80.0, 60.0);
painter = QPainter(self);
painter.drawRect(rectangle);
```

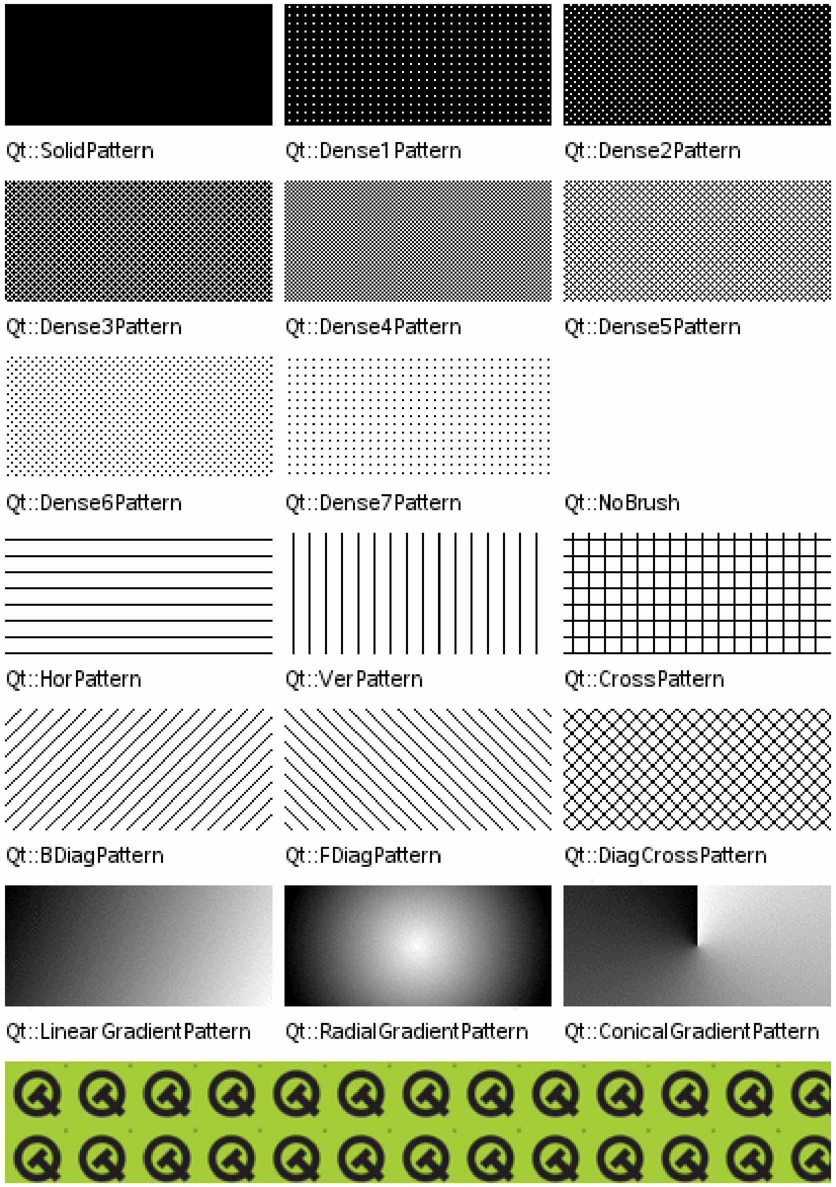


```
rectangle = QRectF(10.0, 20.0, 80.0, 60.0);
painter = QPainter(self);
painter.drawRoundRect(rectangle);
```

Brush

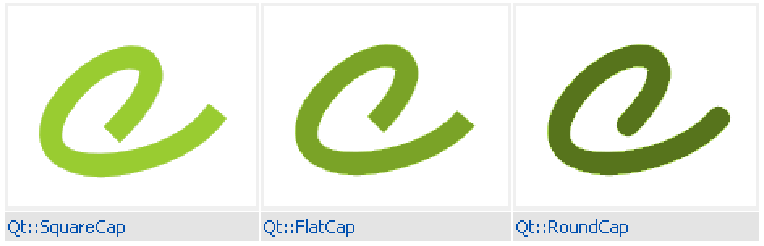
- Defines the fill pattern for shapes
- Styles (mutually exclusive)
  - Pattern
    - Color (Qt predefined color or custom QColor)
  - Gradient
    - Substyle (linear, radial, conical) / color
  - Texture
    - Image

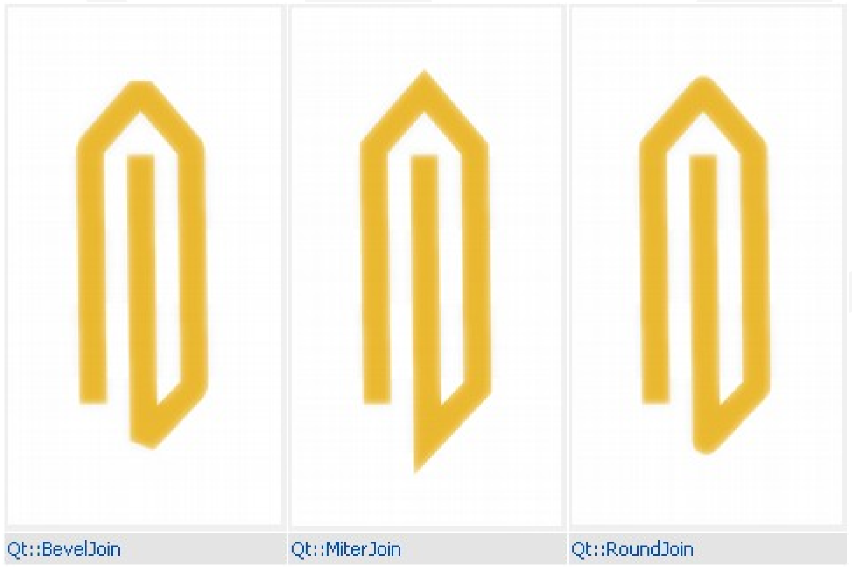
Brush styles



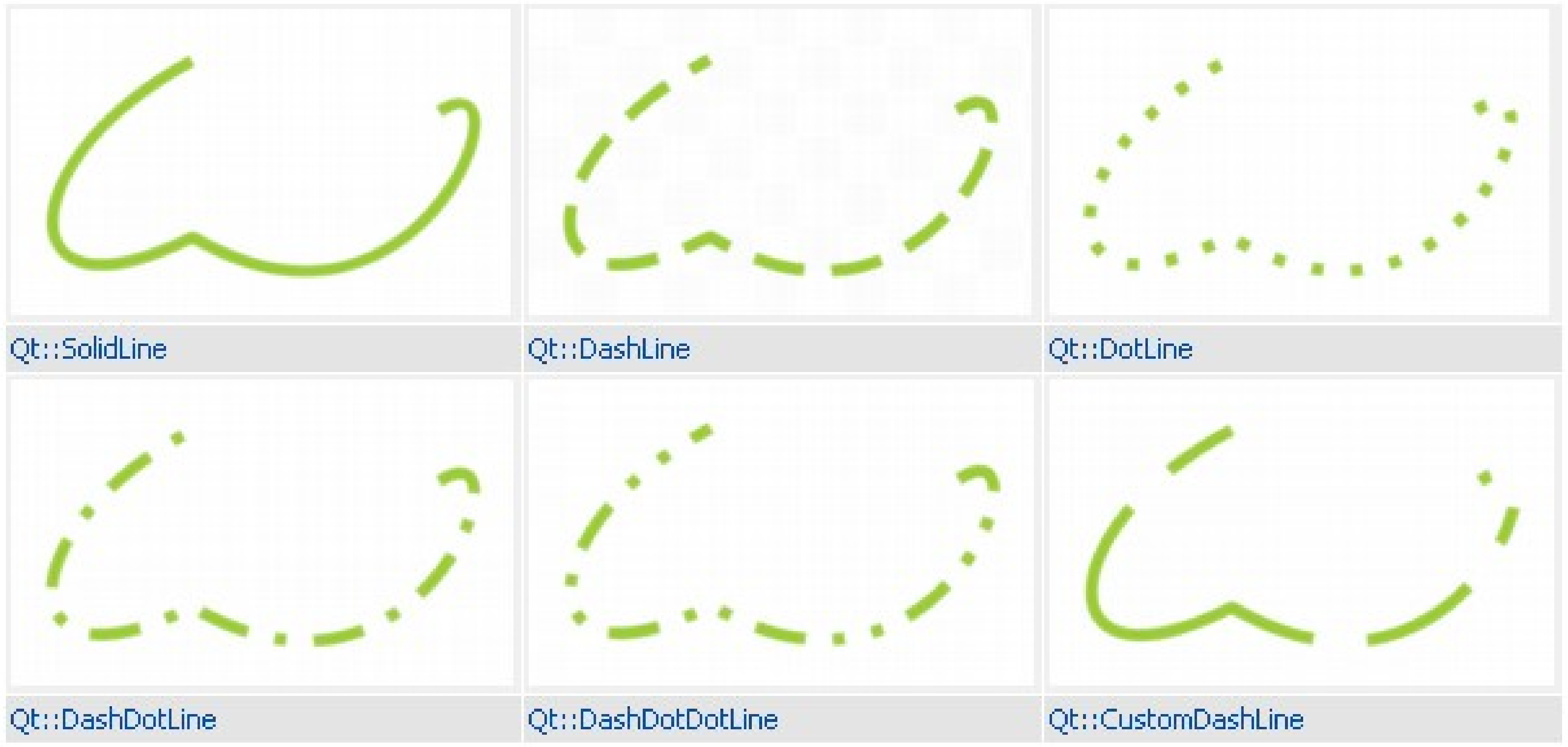
Pen

- Defines the color and stipple pattern used to draw lines and boundaries
- Can have a brush, to fill the strokes
- Boundary styles (cap style and join style)





Pen styles


Qt::SolidLine
Qt::DashLine
Qt::DotLine
Qt::DashDotLine
Qt::DashDotDotLine
Qt::CustomDashLine

Draw your UI

Custom Widgets & QPainter

Track your Location!

```
class TrackingArea(QWidget):
    def __init__(self, parent=None):
        super(TrackingArea, self).__init(self, parent)
        self.setMouseTracking(True)

    # QMouseEvent: Mouse Handling (click, move, ...)
    def mousePressEvent(self, event):
        print 'Mouse Press', event.pos()

    def mouseReleaseEvent(self, event):
        print 'Mouse Release', event.pos()

    def mouseMoveEvent(self, event):
        print 'Mouse Move', event.pos()

    # QKeyEvent: Keyboard Handling (modifiers, key, ...)
    def keyPressEvent(self, event):
        print 'Key Press', event.key(), event.text()
```

QtWebkit

Interact with the Web!

A Bridge between Web & Desktop

With QtWebKit you can

- (easily!) embed a fully functional, standard compliant, web browser inside your application
- inspect/extract the content
- manipulate the web page
- rendering web pages on different devices (image, printer, ...)

WebKit is an open source state of the art rendering engine

NOTE: JavaScriptCore is used as JS Engine, check QTWEBKIT-258 for v8 support...

Display a WebPage in 3 lines

webkit/hello_browser.py

```
from PyQt4.QtWebKit import *
from PyQt4.Qt import *
import sys

if __name__ == '__main__':
    if len(sys.argv) < 2:
        print "Usage: simple_browser <url>"
        sys.exit(1)

    app = QApplication([])

    view = QWebView()
    view.load(QUrl(sys.argv[1]))
    view.show()

    app.exec_()
```

...closer to a real Browser

webkit/usable_browser.py

```
class BrowserView(QWidget):
    def __init__(self, parent=None):
        super(BrowserView, self).__init__(parent)

        vlayout = QVBoxLayout()
        self.setLayout(vlayout)

        self.urledit = QLineEdit()
        vlayout.addWidget(self.urledit)

        self.webview = QWebView()
        vlayout.addWidget(self.webview, stretch=1)

        QObject.connect(self.urledit,
                    SIGNAL('returnPressed()'), self._loadUrl)

    def _loadUrl(self):
        self.webview.load(QUrl(self.urledit.text()))
```

Event Loop & WebPage load!

webkit/web_screenshot.py

Load WebPage in a sync way.

```
def _loadWebPage(url):
    eloop = QEventLoop()
    load_finished = []

    def _loadFinished(ok):
        load_finished.append(ok)
        eloop.quit()

    page = QWebPage()
    main_frame = page.mainFrame()
    main_frame.setScrollBarPolicy(Qt.Vertical, Qt.ScrollBarAlwaysOff)
    main_frame.setScrollBarPolicy(Qt.Horizontal, Qt.ScrollBarAlwaysOff)
    QObject.connect(page, SIGNAL('loadFinished(bool)'), _loadFinished)
    main_frame.load(url)

    if not load_finished:
        eloop.exec_()

    return page
```

Take a Web shot!

webkit/web_screenshot.py

```
def _webScreenshot(url):
    page = _loadWebPage(url)
    main_frame = page.mainFrame()

    size = main_frame.contentsSize()
    size = QSize(max(size.width(), 800), min(size.height(), 2048))
    page.setViewportSize(size)

    image = QImage(size, QImage.Format_ARGB32_Premultiplied)
    painter = QPainter(image)
    main_frame.render(painter)
    painter.end()

    return image
```

QtOpenGL

Easy to use OpenGL in Qt applications

Qt and OpenGL

- QGLWidget: a more direct approach to OpenGL rendering
- You have a choice between drawing with QPainter and direct GL commands
- Qt does not have an in-house implementation of OpenGL: the system one will be used.

QGLWidget

- QGLWidget is a widget for rendering OpenGL graphics and integrating it into a Qt application
  - Its associated QPaintEngine uses OpenGL
  - All QPainter drawing primitives are internally translated by the engine to OpenGL commands
  - You can get 2D rendering accelerated via OpenGL simply by using a QGLWidget instead of a QWidget and redefining QGLWidget.paintEvent

QGLWidget revisited

- Receives paint events like normal QWidgets
- QGLWidget.paintEvent must not be redefined
- Three convenient methods exist
  - initializeGL
  - resizeGL
  - paintGL
- Convenience methods qglClearColor, qglColor

QGLWidget revisited

- initializeGL is called just once, immediately before a resizeGL/paintGL sequence
  - first-time initialization goes here

```
def initializeGL(self):
    qglClearColor(Qt.black)
    glShadeModel(GL_FLAT)
    glEnable(GL_DEPTH_TEST)
```

QGLWidget revisited

- resizeGL immediately follows an initializeGL, and is also called if the widget is resized

```
def resizeGL(width, height):
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    ar = width / height
    glFrustum(-ar, ar, -1.0, 1.0, 4.0, 45.0)
    glMatrixMode(GL_MODELVIEW)
```

QGLWidget revisited

- paintGL is called everytime the widget needs to be redrawn

```
def paintGL(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    # glTranslate/glRotate/...
    glBegin(GL_TRIANGLES)
    # qglColor(QColor(...))
    # glVertex3f...
    glEnd()
```

QGLWidget revisited

- Everything seen is standard OpenGL
  - Except for the qglClearColor and qglColor helpers, but you can use the common glColor* calls instead
- Since QGLWidget is a QWidget, it is also possible to redefine custom event callbacks to handle mouse, keyboard, etc...

QGLWidget revisited

- In application with more than one QGLWidget, it is possible to share OpenGL display lists between GL contexts

```
w1 = QGLWidget(self)

w2 = QGLWidget(self, w1)
```

- w2 will share its display lists with w1
  - No overhead, the implementation simply shares OpenGL ids

Overpainting

- Overpainting is an approach that allows to use a QGLWidget both as a 3D view with OpenGL and a 2D view with QPainter.
- Both 2D and 3D drawing work on the same GL context
- We redefine initializeGL, resizeGL and paintEvent directly (not paintGL)

Overpainting (live demo)

- QGLWidget.paintEvent
  - Push GL attributes and matrices
  - Perform typical initializeGL operations
  - Perform typical resizeGL operations
  - Draw the 3D scene
  - Pop GL attributes and matrices
  - Create and “begin” a QPainter
  - Draw the 2D scene overpainting with QPainter
  - “End” the QPainter

Mix QPainter and native OpenGL

- OpenGL is a giant state machine
- Avoid getting in the way of the underlying OpenGL Qt paint engine
- Since Qt 4.6:
    - QPainter.beginNativePainting()
    - QPainter.endNativePainting()

QtDeclarative

Qt Quick & QML

Qt Quick

- Technology to build slick UIs
- Built on Qt technology stack
- Qt Quick = QML + tools

QML

- Declarative language to describe UIs
- Visual editor available (1:1 connection)
- Integration with PyQt
- “Pure” applications (qmlviewer mode)
- “Hybrid” applications
    - QObject slots can be called from QML
    - QObject prop changes are notified to QML

How to use

```
# Create the QML user interface.
view = QDeclarativeView()
view.setSource(QUrl('app.qml'))

# Set to size of the view
view.setResizeMode(QDeclarativeView.SizeRootObjectToView)

# Show the QML user interface.
view.show()
```

Internationalization with Qt

Making the application usable
by people in countries other than one's own.

What is i18n about?

- Embracing a specific national environment:
    - Language
    - Line break behaviour
    - Writing direction
    - Conventions
- ...without hampering development

(True) Horror Story

We're in 2002, and a big Italian company wants to localize their CAD program in Spanish.

If only it wasn't for...

(True) Horror Story

- Strings were initially hardcoded in Italian, English was retrofitted at some point...
- ...with lots of if/else statements
- First try: add another else branch for each string in code (...)
- Second try: tool to produce multiple codebases – one for each language (...)
- AFAIK still unfinished two years later

All we need is a good workflow

- Developers produce i18n-ready code
    - With no codebase pollution
- Translators translate strings
    - Iteratively (code and strings can change!)
    - No technical knowledge needed
- The framework does the rest

Developers' step 1

- **QObject.tr()**
  - Parse-time: marks strings
  - Run-time: translates strings
- Not everything is a QObject...
  - QCoreApplication.translate()
  - QtCore.QT_TR_NOOP()

Developers' step 2

- **Use QString for all user visible text**
- QString are Unicode strings → transparent processing of strings (reg exp, split etc)

Developers' step 3

- **Use QString.arg() for dynamic text**
- QString.arg() allows easy argument reordering

```
def showProgress(self done, total, current_file):

     label.setText(self.tr("%1 of %2 files copied.\nCopying: %3")
           .arg(done)
           .arg(total)
           .arg(currentFile))
```

Some glue

- Add a TRANSLATIONS entry to .pro
- Run `pylupdate4` to extract a .ts file
- Send .ts file to translators
- Run `lrelease` to produce translated binary files
- Set up a `QTranslator`
- `QCoreApplication.installTranslator()`

Translators' (only) step

- Open .ts file with Linguist
- Fill the missing translations
- There is no step 3

- Developer: “...hey that's not fair!”

Some case studies

- Nearing the 2.0 release:
  - Parse again with *pylupdate4*
  - Fill only the missing translations
- Wanting to add a language
  - Add that language to TRANSLATIONS
  - Run pylupdate4, translate the .ts file, lrelease
  - This time it's a fairy tale!
- Update language on the fly
  - installTranslator sends a changeEvent

Gotchas

- ::tr assumes latin-1
- What about Designer files?

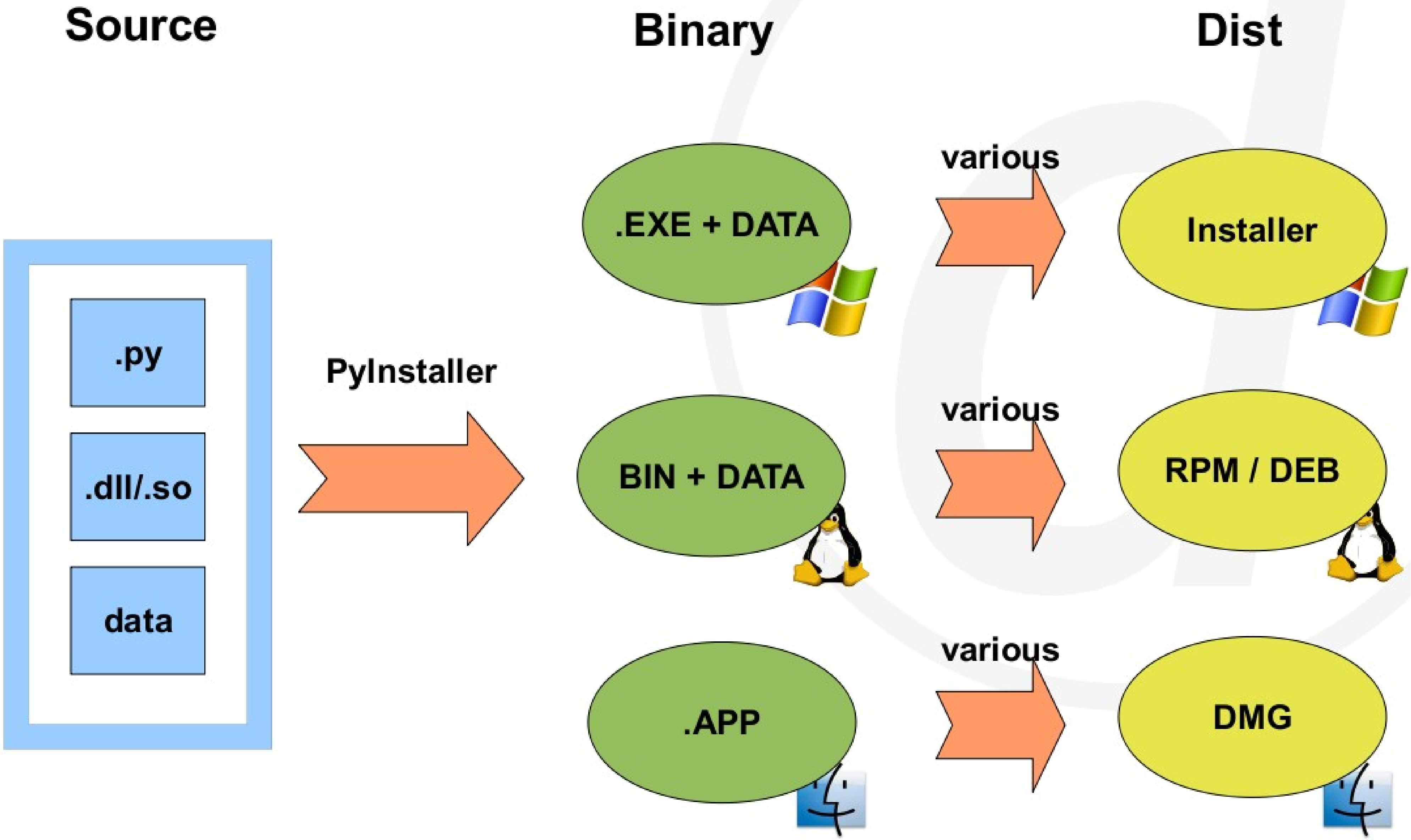
PyInstaller

Distribute your Python programs
as a stand-alone executables

Wrapping things up

- PyQt programs are often composed of:
  - Python source
  - PyQt libraries (.dll or .so)
  - Data files
- How to distribute them?
  - “Customer please apt-get python and pyqt”
  - “What is apt-get?”

Overview



. software . hardware . innovation

develer

PyInstaller crash course

- Get PyInstaller 1.5
- python Configure.py
- python Makespec.py <program>.py
- python Build.py <program>.spec
- ./dist/program/program

Feature reel

- Free as in beer and freedom
- Multiplatform
    - PyInstaller exclusive
- Built-in support for 3rd party libraries
    - PyInstaller exclusive
- Compression (upx)

Under the hood

- SPEC file: PyInstaller project (in Python)
  - one-file / one-dir modes
  - windowed / console
  - debug
  - icon, verison, etc...

Dependencies

- Entry-point module in Analysis call
- Recursively analyze bytecode
    - Explicit imports
    - ctypes LoadLibrary
- Hidden imports
    - Library-specific hooks

“Wait, I'm still giving my source away!”

- No source code
    - Still, bytecode can be extracted
- Crypt support
    - Custom code needed for this

GRAZIE !

Develer S.r.l.
Via Mugellese 1/A
50013 Campi Bisenzio
Firenze - Italia

Contatti

Mail: info@develer.com

Phone: +39-055-3984627

Fax: +39 178 6003614

http://www.develer.com