12th Summer School on SCIENTIFIC VISUALIZATION

Introduction to GUI programming in Python

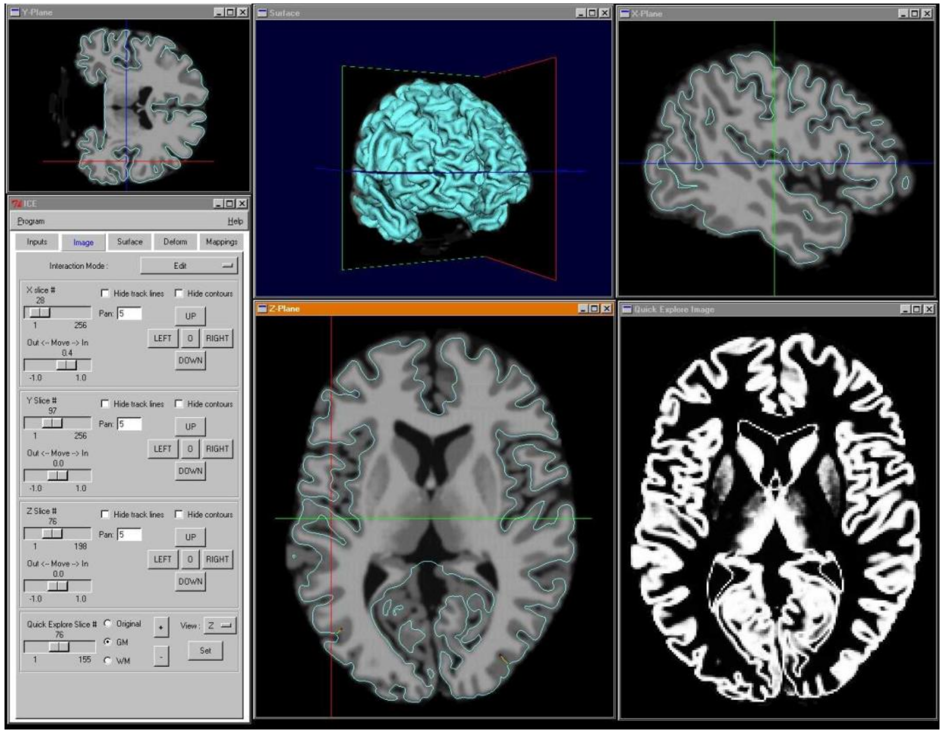
Alice Invernizzi
a.invernizzi@cineca.it

CINECA

Index

Introduction to GUI

- GUI (Graphical User Interface) is a type of interface that allows users to communicate with eletronic devices using images rather than text command.
- A GUI represents the information and actions available to a user through graphical icons. The actions are usually performed through direct manipulation of the graphical elements.



Introduction to GUI

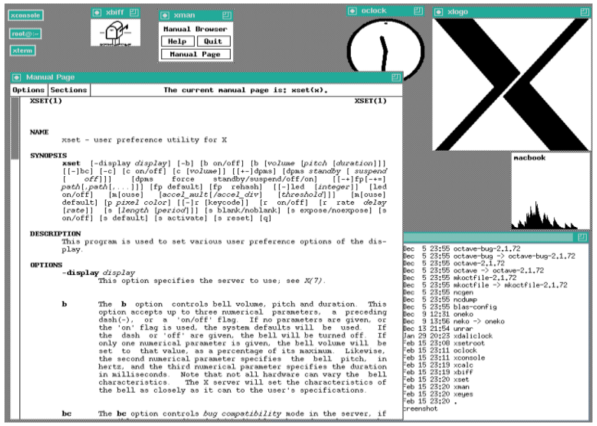
The precursor to GUIs was invented by researchers at the Stanford Research Institute, led by Douglas Engelbart. They developed the use of text-based hyperlinks manipulated with a mouse (1963)

In 1983, the Apple Lisa was first GUI offering.

Introduction to GUI

The X Windows System was introduced in the mid-1980s to provide graphical support for unix operating systems.

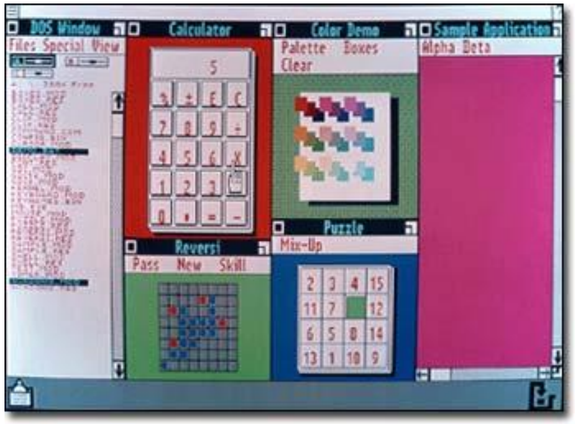
Microsoft introduced
A Windows 1.0 in
1985



The current manual page is: xset(x).

```
XSET(1)                                                    XSET(1)

NAME
       xset - user preference utility for X

SYNOPSIS
       xset  [-display display] [-b] [b on/off] [b [volume [pitch [duration]]]
       [[-]bc] [-c] [c on/off] [c [volume]] [[+-]dpms] [dpms standby [ suspend
       [   off]]]   [dpms   force   standby/suspend/off/on]   [[-+]fp[-+=]
       path[,path[,...]]] [fp default] [fp  rehash]  [[-]led  [integer]]  [led
       on/off]   [m[ouse]   [accel_mult[/accel_div]   [threshold]]]   [m[ouse]
       default] [p pixel color] [[-]r [keycode]]  [r  on/off]  [r  rate  delay
       [rate]]  [s [length [period]]] [s blank/noblank] [s expose/noexpose] [s
       on/off] [s default] [s activate] [s reset] [q]

DESCRIPTION
       This program is used to set various user preference options of the dis-
       play.

OPTIONS
       -display display
              This option specifies the server to use; see X(7).

       b      The  b  option  controls bell volume, pitch and duration.  This
              option accepts up to three numerical  parameters,  a  preceding
              dash(-),  or  a  'on/off' flag.  If no parameters are given, or
              the 'on' flag is used, the system defaults will  be  used.   If
              the  dash  or 'off' are given, the bell will be turned off.  If
              only one numerical parameter is given, the bell volume will  be
              set  to  that value, as a percentage of its maximum.  Likewise,
              the second numerical parameter specifies  the  bell  pitch,  in
              hertz, and the third numerical parameter specifies the duration
              in milliseconds.  Note that not all hardware can vary the  bell
              characteristics.   The X server will set the characteristics of
              the bell as closely as it can to the user's specifications.

       bc     The bc option controls bug compatibility mode in the server, if
```

```
Dec  5 23:55 octave-bug-2.1.72
Dec  5 23:55 octave-bug -> octave-bug-2.1.72
Dec  5 23:55 octave-2.1.72
Dec  5 23:55 octave -> octave-2.1.72
Dec  5 23:55 mkoctfile-2.1.72
Dec  5 23:55 mkoctfile -> mkoctfile-2.1.72
Dec  5 23:55 ncgen
Dec  5 23:55 ncdump
Dec  5 23:55 blas-config
Dec  9 12:31 oneko
Dec  9 13:56 neko -> oneko
Dec 13 21:54 unrar
Jan 29 20:23 xdaliclock
Feb 15 23:08 xsetroot
Feb 15 23:11 oclock
Feb 15 23:11 xconsole
Feb 15 23:19 xcalc
Feb 15 23:19 xbiff
Feb 15 23:20 xset
Feb 15 23:20 xman
Feb 15 23:20 xeyes
Feb 15 23:20 .
creenshot
```



The GUIs familiar to most people today are Microsoft Windows, Mac OS X, and the X Window System interfaces for desktop and laptop computers, and Symbian, BlackBerry OS, Android, Windows Phone, and Apple's iOS for handheld ("smartphone") devices.

GUI programming in Python

Python has a huge number of GUI frameworks (or toolkits) available for it,from Tkinter (traditionally bundled with Python, using Tk) to a number of other cross-platform solutions, as well as bindings to platform-specific technologies.

EasyGui: is a module for very simple, very easy GUI programming in Python.
Tkinter: standard GUI toolkit included with Python, simple and easy
WxPython: xWidgets is a C++ library that lets developers create applications for Windows, OS X, Linux and UNIX, with binding for Python
PyQt: Python bindings for the Qt application development framework , not just GUI features

For a complete list:
http://wiki.python.org/moin/GuiProgramming

Introduction to Qt

What is Qt?
Qt is a cross platform development framework written in C++.

❑Qt was developed by Trolltech (now owned by Nokia)
❑ Though written in C++, Qt can also be used in several other programming languages, through language bindings available for Ruby, Java, Perl, and also Python with PyQt.
❑The Qt toolkit is a collection of classes to simplify the creation of programs. Qt is more than just a GUI toolkit:
Databases, XML, WebKit, multimedia, networking, OpenGL, scripting, non-GUI..
❑ Qt is available on several platforms, in particular: Unix/Linux, Windows, Mac OS

Introduction to Qt

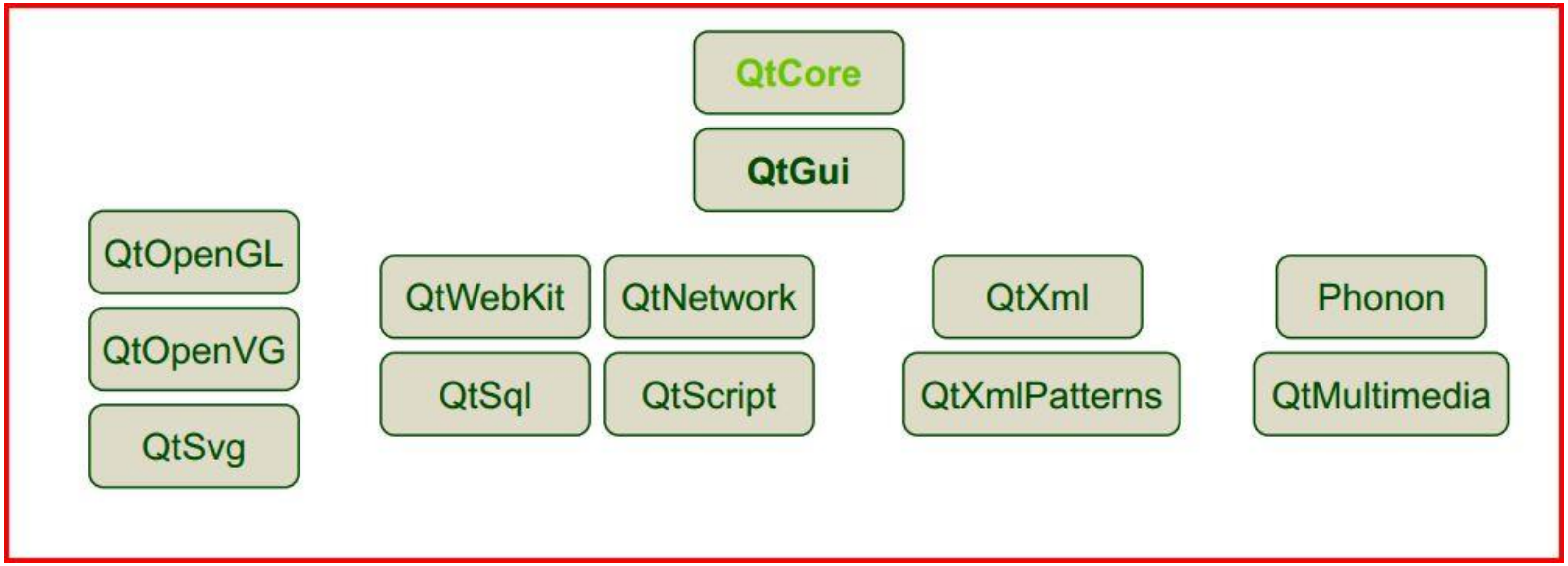
Code less.
Create more.
Deploy everywhere.

Code less because even the function linking is drag and drop capable!
Create more because you spend less time coding and more time innovating!
Lastly, **Deploy Everywhere** because it CAN run on any of its supported platforms (Windows, supported Linux, Mac, supported Symbian) without altering the code

Introduction to Qt

Qt is built from modules

¤ All modules have a common scheme and are built from the same API design ideas



Introduction to Qt

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

Introduction to Qt

QtGUI

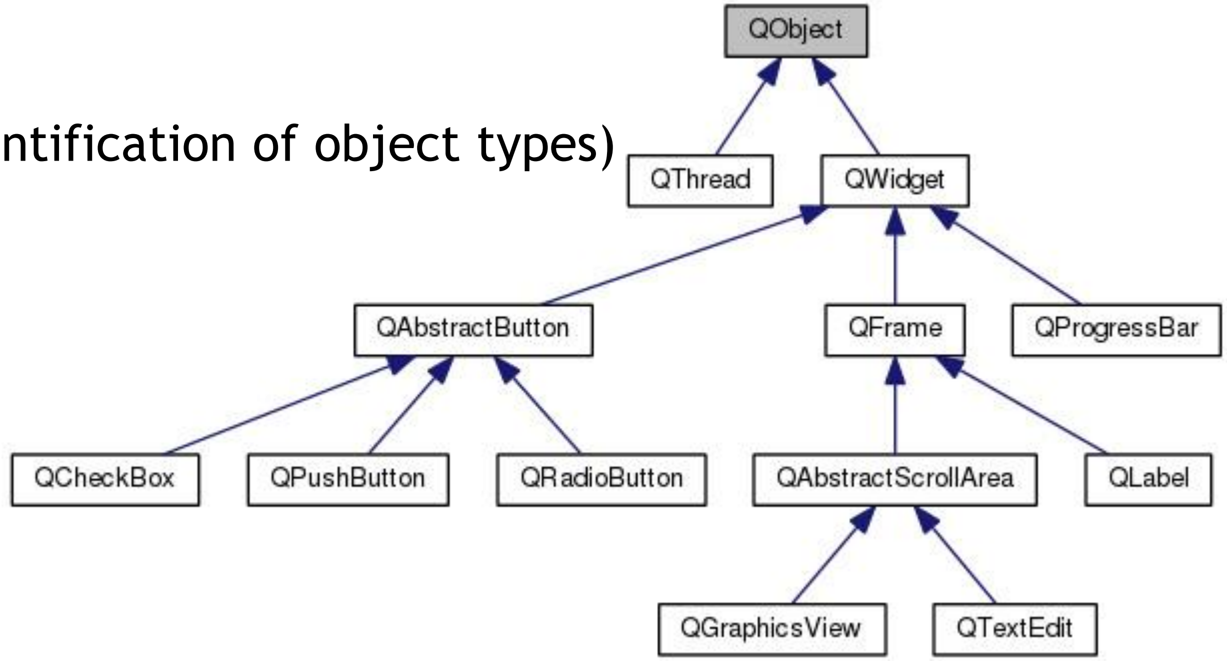
- Widgets:
QCheckBox, QComboBox, QDateTimeEdit, QLineEdit, QPushButton,QRadioButton, QSlider, QSpinBox, etc.
- Basic value types:
QColor, QFont, QBrush, QPen
- Painting system and devices:
QPainter, QPaintDevice, QPrinter, QImage, QPixmap, QWidget
- Basic application support:
QApplication – encapsulates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

QObject Model

The QObject class is the base class of all Qt objects.

QObject is the heart of the Qt Object Model.
Three major responsibilities of QObject:

- Memory Management
- Introspection (runtime identification of object types)
- Event handling

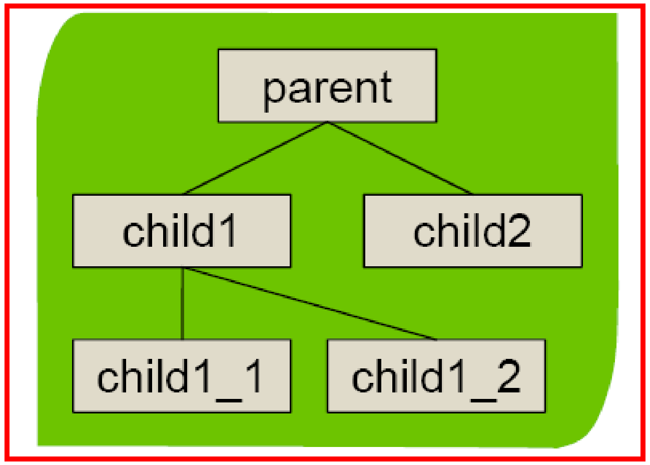


Qt Object Model

All PyQt classes that derive from QObject can have a “parent”.
A widget that has no parent is a top-level window, and a widget that has a parent (always another widget) is contained (displayed) within its parent.

PyQt uses the parent-child ownership model to ensure that if a parent—for example, a top-level window—is deleted, all its children, for example, all the widgets the window contains, are automatically deleted as well

Object Ownership

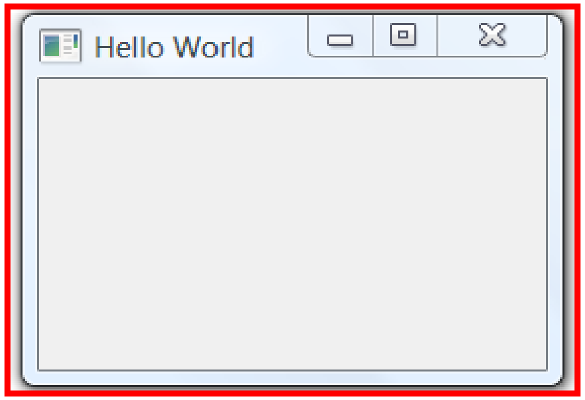


PyQt Simple Example

A tiny PyQt applications has the following elements:

- an application object
- a main window (which has a central widget), or
- a main widget

This is the traditional "Hello World" application, with as little code as possible:



PyQt Simple Example

The Basic GUI widget are located in QtGui module

```
import sys
from PyQt4 import QtGui
app = QtGui.QApplication(sys.argv)
widget = QtGui.QWidget()
widget.resize(250, 150)
widget.setWindowTitle('Hello World')
widget.show()
sys.exit(app.exec_())
```

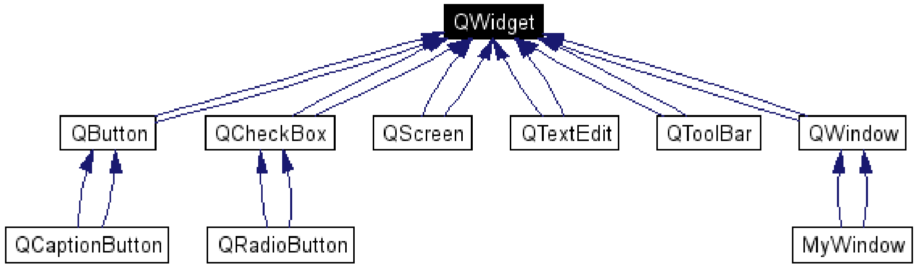
Every PyQt4 application must define An application object located in QtGui

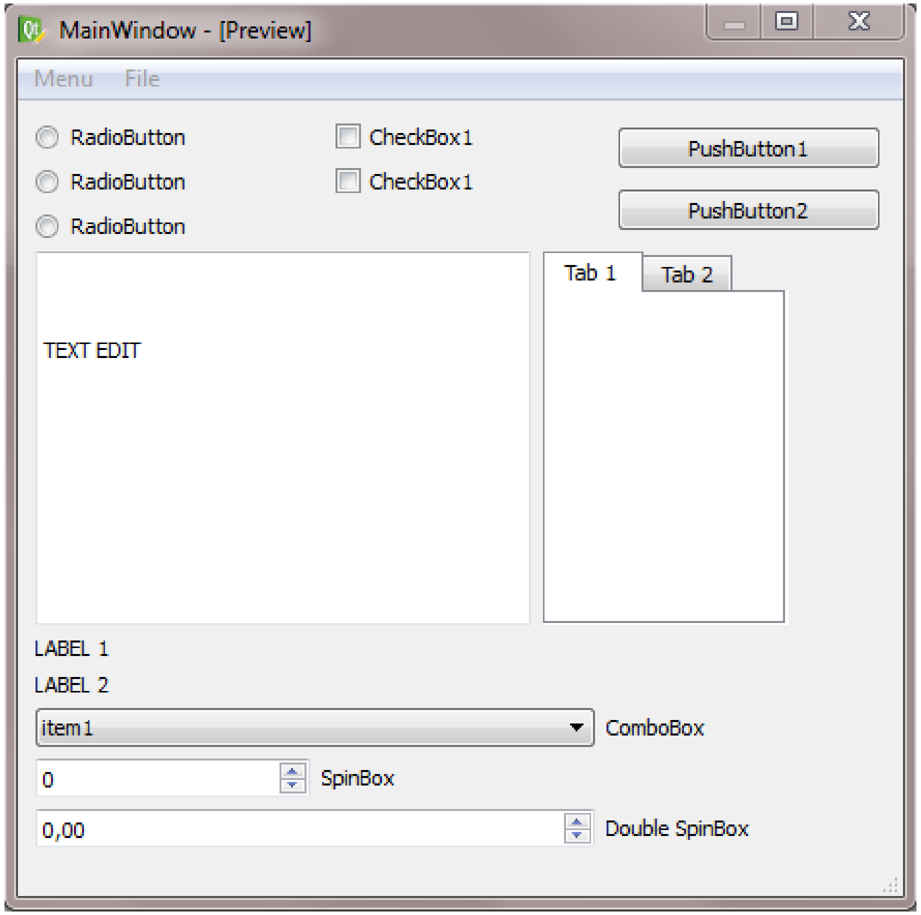
The QWidget widget is the base class of all user interface objects in PyQt4

Finally, we enter the mainloop of the application. The event handling starts from this point. The mainloop receives events from the window system and dispatches them to the application widgets. The mainloop ends, if we call the exit() method or the main widget is destroyed

GUI Components

User interfaces are built from individual widgets
There are more than 59+ direct descendants from Qwidget.





Simple PyQt Example

OOP style. QtFramework is an OO framework

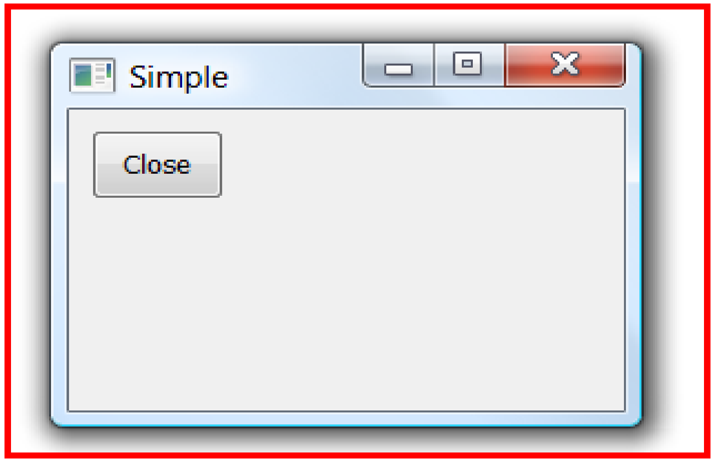
```
import sys
from PyQt4 import QtGui, QtCore
class QuitButton(QtGui.QWidget):
  def __init__(self, parent=None):
      QtGui.QWidget.__init__(self, parent)
      self.setGeometry(300, 300, 250, 150)
      self.setWindowTitle('Simple')
      quit = QtGui.QPushButton('Close', self)
      quit.setGeometry(10, 10, 60, 35)
      self.connect(quit, QtCore.SIGNAL('clicked()'),QtGui.qApp,
QtCore.SLOT('quit()'))
app = QtGui.QApplication(sys.argv)
qb = QuitButton()
qb.show()
sys.exit(app.exec_())
```

Ownership, parent-child

Simple PyQt Example

```
import sys
from PyQt4 import QtGui, QtCore
app = QtGui.QApplication(sys.argv)
widget=QtGui.QWidget()
widget.setWindowTitle('Simple')
widget.setGeometry(300, 300, 250, 150)
button=QtGui.QPushButton('Close',widget)
button.setGeometry(10, 10, 60, 35)
app.connect(button, QtCore.SIGNAL('clicked()'),QtGui.qApp, QtCore.SLOT('quit()'))
widget.show()
sys.exit(app.exec_())
```

Connection to manage GUI events



Layout Management

Important thing in programming is the layout management. Layout management is the way how we place the widgets on the window. The management can be done in two ways. We can use absolute positioning or layout classes.

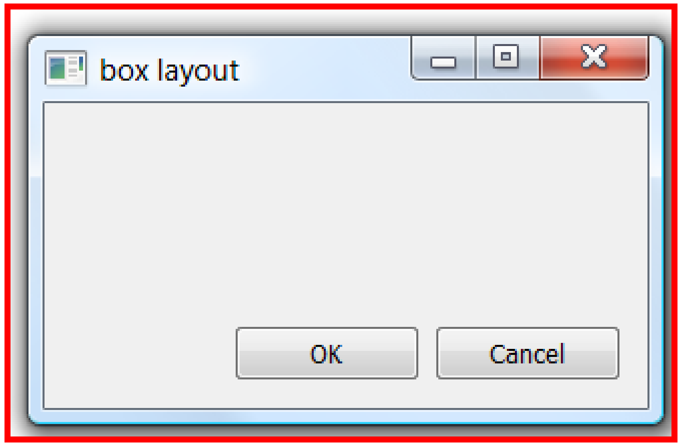
The programmer specifies the position and the size of each widget in pixels.

NOTES:

- The size and the position of a widget do not change, if you resize a window
- Applications might look different on various platforms

Box Layout

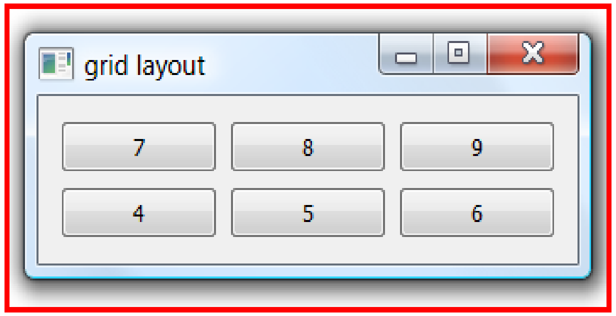
Layout management with layout classes is much more flexible and practical. It is the preferred way to place widgets on a window. The basic layout classes are QHBoxLayout and QVBoxLayout.



```
class BoxLayout(QtGui.QWidget):
def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle('box layout')
        ok = QtGui.QPushButton("OK")
        cancel = QtGui.QPushButton("Cancel")
        hbox = QtGui.QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(ok)
        hbox.addWidget(cancel)
        vbox = QtGui.QVBoxLayout()
        vbox.addStretch(1)
        vbox.addLayout(hbox)
        self.setLayout(vbox)
        self.resize(300, 150)
app = QtGui.QApplication(sys.argv)
qb = BoxLayout()
qb.show()
sys.exit(app.exec_())
```

Grid Layout

The most universal layout class is the grid layout. This layout divides the space into rows and columns.

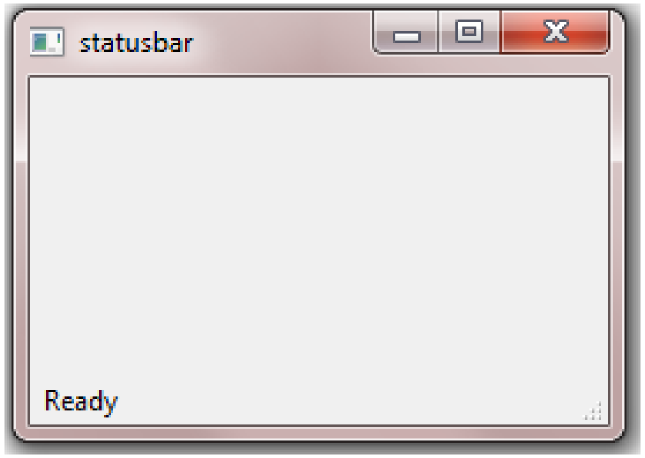


```
import sys
from PyQt4 import QtGUi
class GridLayout(QtGui.QWidget):
def __init__(self, parent=None):
     QtGui.QWidget.__init__(self, parent)
                    self.setWindowTitle('grid layout')
     names = [ '7', '8', '9', '4', '5', '6']
     grid = QtGui.QGridLayout()
     j = 0
     pos = [(0, 0), (0, 1), (0, 2), (1, 0), (1,1),(1,2)]
     for i in names:
           button = QtGui.QPushButton(i)
           grid.addWidget(button, pos[j][0], pos[j][1])
           j = j + 1
     self.setLayout(grid)
app = QtGui.QApplication(sys.argv)
qb = GridLayout()
qb.show()
sys.exit(app.exec_())
```

Main Window

The QMainWindow class provides a main application window. This enables to create the classic application skeleton with a statusbar, toolbars and a menubar.

The `statusbar` is a widget that is used for displaying status information.



```
import sys
from PyQt4 import QtGui
class MainWindow(QtGui.QMainWindow):
 def __init__(self):
        QtGui.QMainWindow.__init__(self)
        self.resize(250, 150)
        self.setWindowTitle('statusbar')
        self.statusBar().showMessage('Ready')
app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

Main Window

- A menubar is one of the most visible parts of the GUI application. It is a group of commands located in various menus.
- Toolbars provide a quick access to the most frequently used commands.
- GUI applications are controlled with commands. These commands can be launched from a menu, a context menu, a toolbar or with a shortcut. PyQt simplifies development with the introduction of actions.
- 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

Main Window

QAction

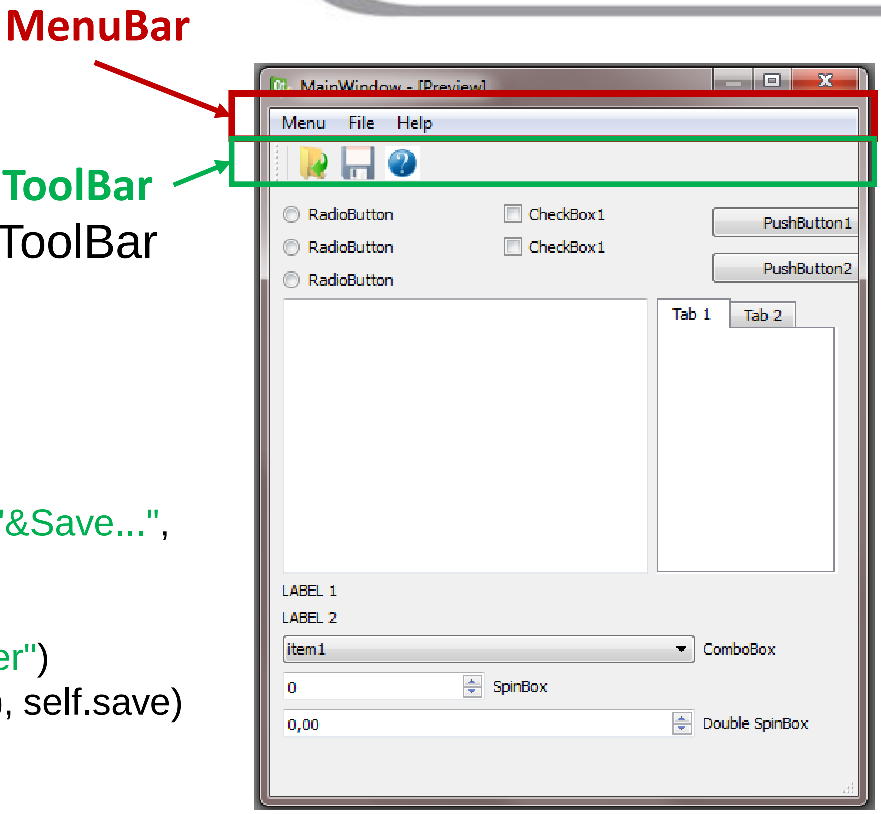
To create an action, you can:

•Instantiate a QAction object directly

•Call addAction() on existing QMenu and QToolBar objects

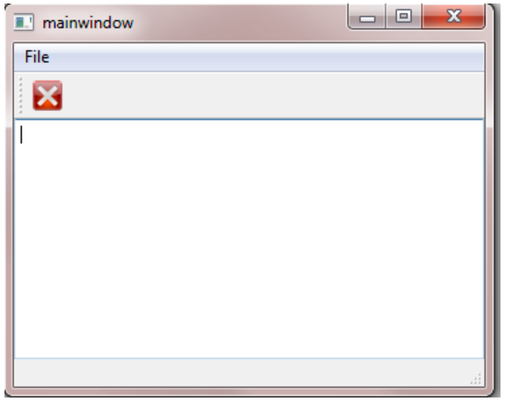
•Then you can share it with other objects.

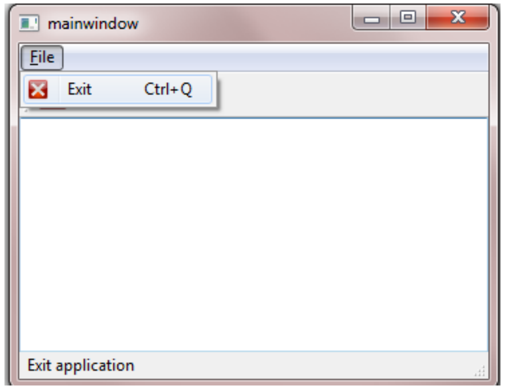
```
self.saveAction = QAction(QIcon(":/images/save.png"), "&Save...",
self)
self.saveAction.setShortcut("Ctrl+S")
self.saveAction.setStatusTip("Save the current form letter")
self.connect(self.saveAct, QtCore.SIGNAL("triggered()"), self.save)
...
self.fileMenu = self.menuBar().addMenu("&File")
self.fileMenu.addAction(self.saveAction)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



Main Window

We will create a menubar, toolbar and a statusbar. We will also create a central widget.





```
import sys
from PyQt4 import QtGui, QtCore
class MainWindow(QtGui.QMainWindow):
def __init__(self):
                QtGui.QMainWindow.__init__(self)
                self.resize(350, 250)
                self.setWindowTitle('mainwindow')
                textEdit = QtGui.QTextEdit()
                self.setCentralWidget(textEdit)
                exit = QtGui.QAction(QtGui.QIcon('\Icon\exit.png'), 'Exit', self)
                exit.setShortcut('Ctrl+Q')
                exit.setStatusTip('Exit application')
                self.connect(exit, QtCore.SIGNAL('triggered()'), QtCore.SLOT('close()'))
                self.statusBar()
                menubar = self.menuBar()
                file = menubar.addMenu('&File')
                file.addAction(exit)
                toolbar = self.addToolBar('Exit')
                toolbar.addAction(exit)
app = QtGui.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```

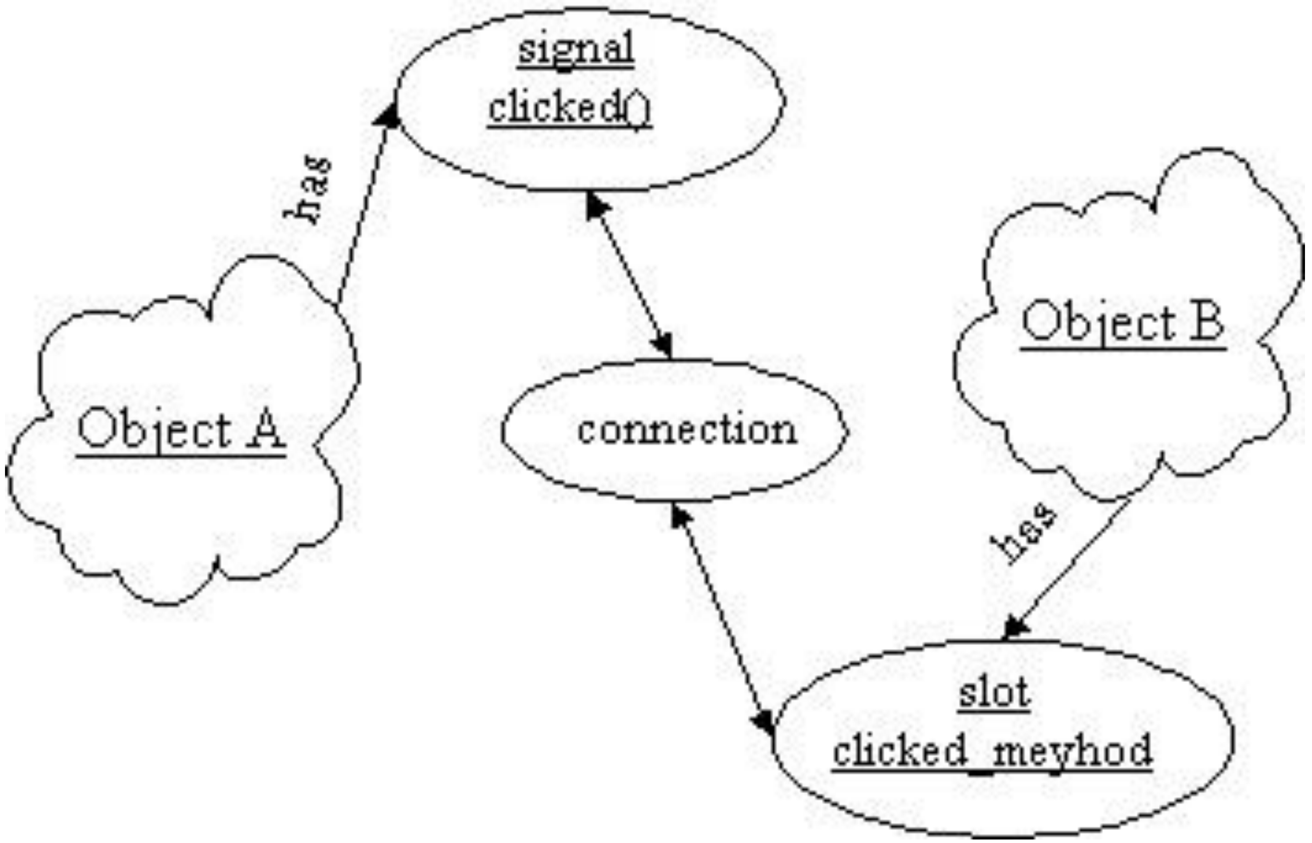
Signal and Slot

Events are an important part in any GUI program. Events are generated by users or by the system. When we call the application's exec_() method, the application enters the main loop. The main loop fetches events and sends them to the objects. Trolltech has introduced a unique signal and slot mechanism.

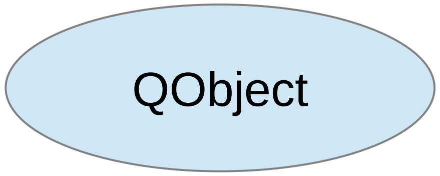
Signals are emitted, when users click on the button, drag a slider etc. Signals can be emitted also by the environment.

A **Slot** is a method, that reacts to a signal. In python, a slot can be any python callable.

Signal and Slot



Signal and Slot



```
Qobject::connect(src,SIGNAL(signature),dest,SLOT(signature))
```

Connections are made using the `connect()` method.

The connect function can take the following parameters:
sender – the QObject that will send the signal.
signal – the signal that must be connected
receiver – the QObject that has the slot method that will be called when the signal is emitted.
slot – the slot method that will be called when the signal is emitted.

Signal and Slot

- In many PyQt code samples the "old-style" signal-slot connection mechanism is used.

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class MyForm(QMainWindow):
def __init__(self, parent=None):
     QMainWindow.__init__(self,parent)
    the_button = QPushButton('Close')
    self.connect(the_button, SIGNAL('clicked()'), self, SLOT('close()'))
    self.setCentralWidget(the_button)

app = QApplication(sys.argv)
form = MyForm()
form.show()
app.exec_()
```

Old-style signal/slot syntax

Signal and Slot

Apart from being verbose and un-Pythonic, this syntax has a serious problem. You must type the C++ signature of the signal exactly. Otherwise, the signal just won’t fire, without an exception or any warning. This is a *very common* mistake.

The "new-style" signal-slot connection mechanism is much better. Here’s how the connection is done

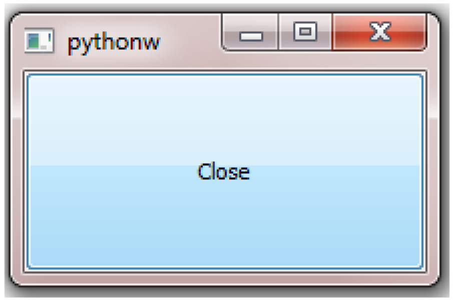
```
sender.signalName.connect(receiver.slotName)
```

This mechanism is supported by PyQt since version 4.5, and provides a safer and much more convenient way to connect signals and slots. Each signal is now an attribute that gets automatically bound once you access it.

Signal and Slot

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
class MyForm(QMainWindow):
def __init__(self, parent=None):
    super(MyForm, self).__init__(parent)
    the_button = QPushButton('Close')
the_button.clicked.connect(self.close)
self.setCentralWidget(the_button)

app = QApplication(sys.argv)
form = MyForm()
form.show()
app.exec_()
```

New-style signal/slot syntax



Signal and Slot

- A signal can be connected to any number of slots. In this case, the slots are activated in arbitrary order.

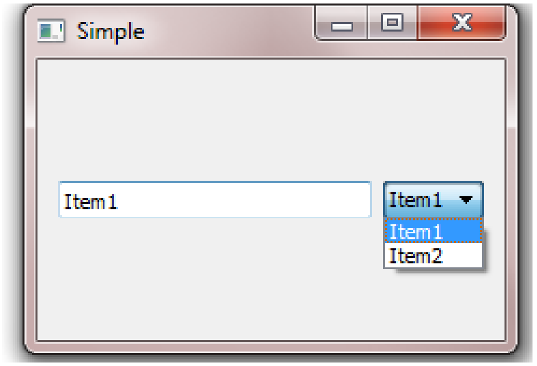
- A slot may not have more arguments than the signal that is connected to it. But may have less; the additional parameters are then discarted

- Corresponding signal and slot arguments must have the same types, so for example,we could not connect a `QDial's valueChanged(int)` signal to a `QLineEdit's setText(QString)` slot.

- A signal may also be connected to another signal.

Signal and Slot

A small example, using built-in signal and slot.

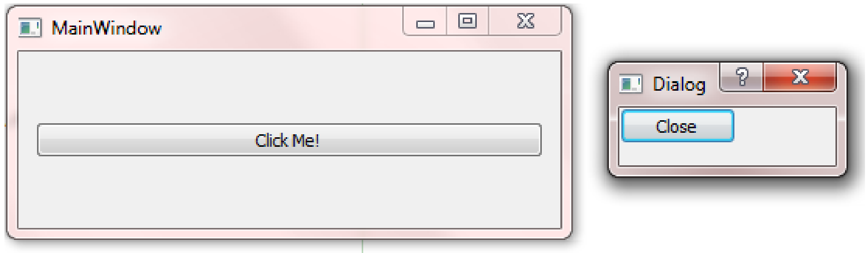


```
from PyQt4 import QtGui, QtCore
import sys
class Test(QtGui.QWidget):
def __init__(self,parent=None):
        QtGui.QWidget.__init__(self, parent)
         self.setWindowTitle('Simple‘)
         self.setGeometry(300,300,250,150)
         self.Line=QtGui.QLineEdit(' ',self)
         self.ComboBox=QtGui.QComboBox(self)
        self.ComboBox.addItem('Item1')
        self.ComboBox.addItem('Item2')
        self.grid=QtGui.QGridLayout()
        self.grid.addWidget(self.Line,0,0)
        self.grid.addWidget(self.ComboBox,0,1)
        self.setLayout(self.grid)
        self.connect(self.ComboBox,QtCore.SIGNAL('currentIndexChanged(QString)'),
        self.Line,QtCore.SLOT('setText(QString)'))
app = QtGui.QApplication(sys.argv)
widget=Test()
widget.setGeometry(300, 300, 250, 150)
widget.show()
sys.exit(app.exec_())
```

Signal and Slot

A small example, using built-in signal and user-defined slot.

```
 from PyQt4 import QtCore, QtGui
 class MyWindow(QtGui.QWidget):
   def __init__(self, parent=None):
       super(MyWindow, self).__init__(parent)
       self.pushButtonWindow = QtGui.QPushButton(self)
       self.pushButtonWindow.setText("Click Me!")
       self.pushButtonWindow.clicked.connect(self.on_pushButton_clicked)
       self.layout = QtGui.QHBoxLayout(self)
       self.layout.addWidget(self.pushButtonWindow)
def on_pushButton_clicked(self):
       s=QtGui.QDialog()
       button=QtGui.QPushButton('Close',s)
       button.clicked.connect(s.close)
       s.exec_()
app = QtGui.QApplication(sys.argv)
app.setApplicationName('MyWindow')
main = MyWindow()
main.show()
sys.exit(app.exec_())
```



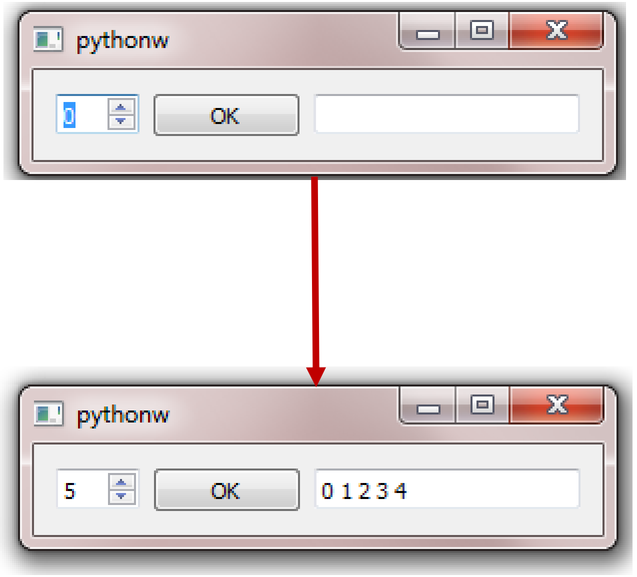
Signal and Slot

A small example, using user-defined signal and user-defined slot.

```
from PyQt4 import QtGui, QtCore
class MyWidget(QtGui.QWidget):
    mysignal = QtCore.pyqtSignal( list)
    def __init__(self, parent=None):
                QtGui.QWidget.__init__(self,parent)
                self.button = QtGui.QPushButton("OK", self)
                self.text=QtGui.QLineEdit()
                self.spin=QtGui.QSpinBox()
                self.grid=QtGui.QGridLayout()
                self.grid.addWidget(self.button,0,1)
                self.grid.addWidget(self.spin,0,0)
                self.grid.addWidget(self.text,0,2)
                self.setLayout(self.grid)
                self.button.clicked.connect(self.OnClicked)
                self.mysignal.connect(self.OnPrint)
```

Signal and Slot

```
    def OnClicked(self):
            val=self.spin.value()
            #self.emit(QtCore.SIGNAL('mysignal'),range(val))
            self.mysignal.emit(range(val))
    def OnPrint(self,val):
            s= ' '
            for el in val:
                s+=str(el)+' '
            self.text.setText(s)
if __name__ == '__main__':
    import sys
    app = QtGui.QApplication(sys.argv)
    w = MyWidget()
    w.show()
    sys.exit(app.exec_())
```



Change Value in SpinBox, press button Ok. LineEdit will be modified depending on the value of SpinBox

Dialogs in PyQt

Dialog windows or dialogs are an indispensable part of most modern GUI applications.

In a computer application a dialog is a window which is used to "talk" to the application. A dialog is used to input data, modify data, change the application settings etc.

Dialogs are important means of communication between a user and a computer program.

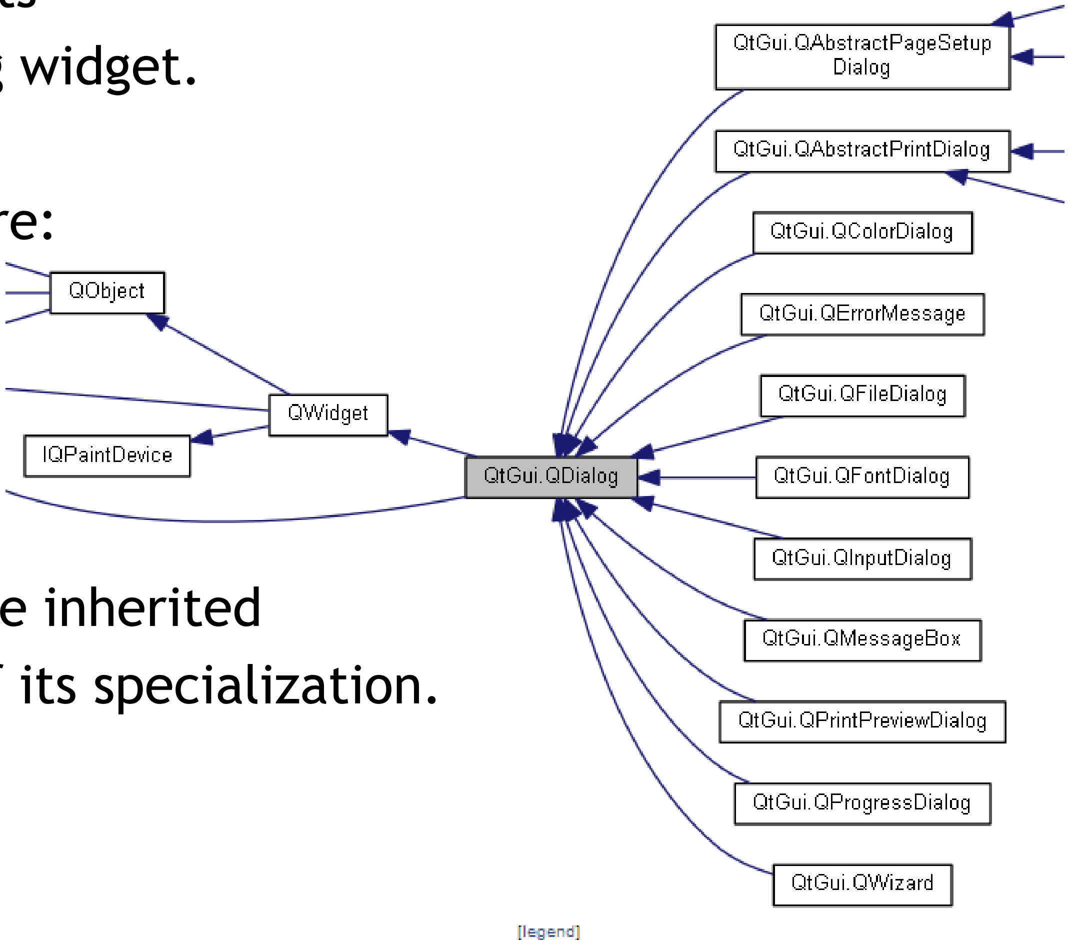
There are essentially two types of dialogs. Predefined dialogs and custom dialogs.

Dialogs in PyQt

- Predefined dialogs inherits from a general base QDialog widget.

- Common dialog widget are:
QFileDialog
QInputDialog
QFontDialog

- Customized widget can be inherited from QDialog or from one of its specialization.

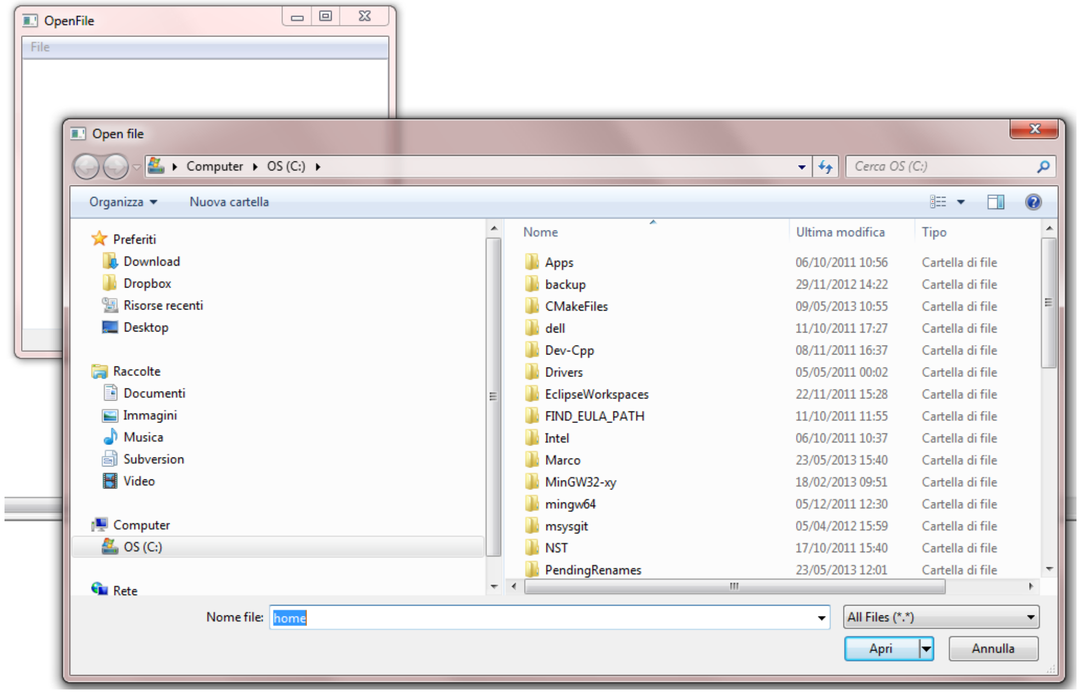


Example: QFileDialog

```
import sys
from PyQt4 import QtGui
from PyQt4 import QtCore
class OpenFile(QtGui.QMainWindow):
   def __init__(self, parent=None):
            QtGui.QMainWindow.__init__(self, parent)
            self.setGeometry(300, 300, 350, 300)
            self.setWindowTitle('OpenFile')
            self.textEdit = QtGui.QTextEdit()
            self.setCentralWidget(self.textEdit)
            self.statusBar()
            self.setFocus()
            exit = QtGui.QAction(QtGui.QIcon('open.png'), 'Open', self)
            exit.setShortcut('Ctrl+O')
            exit.setStatusTip('Open new File')
            self.connect(exit, QtCore.SIGNAL('triggered()'), self.showDialog)
```

Example: QFileDialog

```
        menubar = self.menuBar()
        file = menubar.addMenu('&File')
        file.addAction(exit)
  def showDialog(self):
        filename = QtGui.QFileDialog.getOpenFileName(self, 'Open file','/home')
        file=open(filename)
        data = file.read()
        self.textEdit.setText(data)
app = QtGui.QApplication(sys.argv)
cd = OpenFile()
cd.show()
app.exec_()
```



Nome	Ultima modifica	Tipo
Apps	06/10/2011 10:56	Cartella di file
backup	29/11/2012 14:22	Cartella di file
CMakeFiles	09/05/2013 10:55	Cartella di file
dell	11/10/2011 17:27	Cartella di file
Dev-Cpp	08/11/2011 16:37	Cartella di file
Drivers	05/05/2011 00:02	Cartella di file
EclipseWorkspaces	22/11/2011 15:28	Cartella di file
FIND_EULA_PATH	11/10/2011 11:55	Cartella di file
Intel	06/10/2011 10:37	Cartella di file
Marco	23/05/2013 15:40	Cartella di file
MinGW32-xy	18/02/2013 09:51	Cartella di file
mingw64	05/12/2011 12:30	Cartella di file
msysgit	05/04/2012 15:59	Cartella di file
NST	17/10/2011 15:40	Cartella di file
PendingRenames	23/05/2013 12:01	Cartella di file

Nome file: home

All Files (*.*)

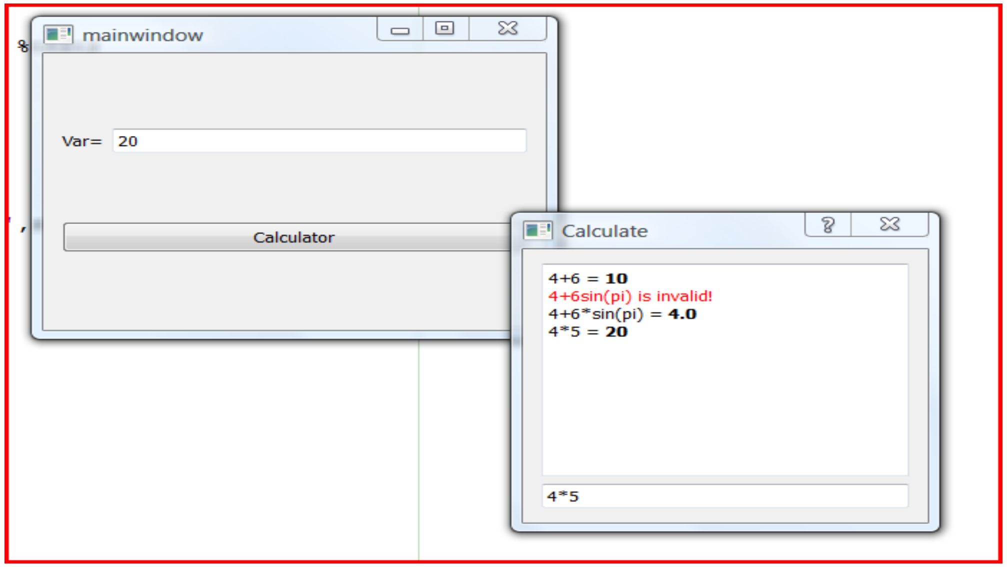
Example: Custom Dialog

```
class Form(QDialog):
    changed=pyqtSignal(QString)
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)
        self.browser = QTextBrowser()
        self.lineedit = QLineEdit("Type an expression and press Enter")
        self.lineedit.selectAll()
        layout = QVBoxLayout()
        layout.addWidget(self.browser)
        layout.addWidget(self.lineedit)
        self.setLayout(layout)
        self.lineedit.setFocus()
        self.connect(self.lineedit,SIGNAL("returnPressed()"),self.updateUi)
        self.setWindowTitle("Calculate")
    def updateUi(self):
        try:
            text = unicode(self.lineedit.text())
            self.browser.append("%s = <b>%s</b>" % (text, eval(text)))
            self.emit(SIGNAL('changed'),unicode(self.lineedit.text())
        except:
            self.browser.append("<font color=red>%s is invalid!</font>" %text)
```

Summer School on SCIENTIFIC VISUALIZATION

Example: Custom Dialog

```
class MainWindow(QtGui.QWidget):
   def __init__(self):
      QtGui.QWidget.__init__(self)
      self.resize(350, 250)
      self.setWindowTitle('mainwindow')
      self.label=QtGui.QLabel('Var=',self)
      self.line=QtGui.QLineEdit('',self)
      self.button=QtGui.QPushButton('Calculator',self)
      self.layoutH = QHBoxLayout()
      self.layoutH.addWidget(self.label)
      self.layoutH.addWidget(self.line)
      self.layoutV = QVBoxLayout()
      self.layoutV.addLayout(self.layoutH)
      self.layoutV.addWidget(self.button)
      self.setLayout(self.layoutV)
      self.button.clicked.connect(self.Dialog)
   def Dialog(self):
      diag=Form()
      diag.changed.connect(self.Update)
      diag.exec_()
   def Update(self,stringa):
      self.line.setText(stringa)
```



```
4+6 = 10
4+6sin(pi) is invalid!
4+6*sin(pi) = 4.0
4*5 = 20
```

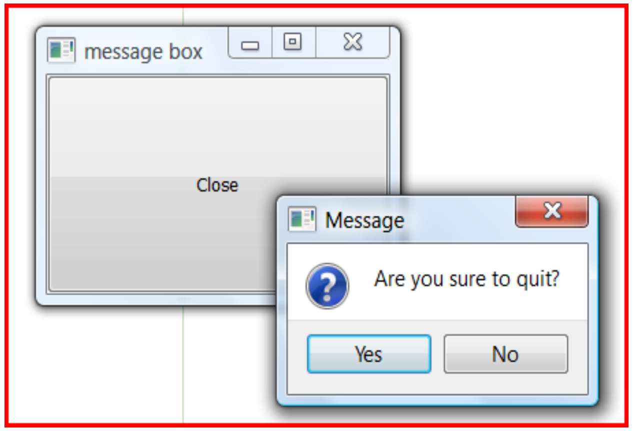
```
4*5
```

CINECA

MessageBox

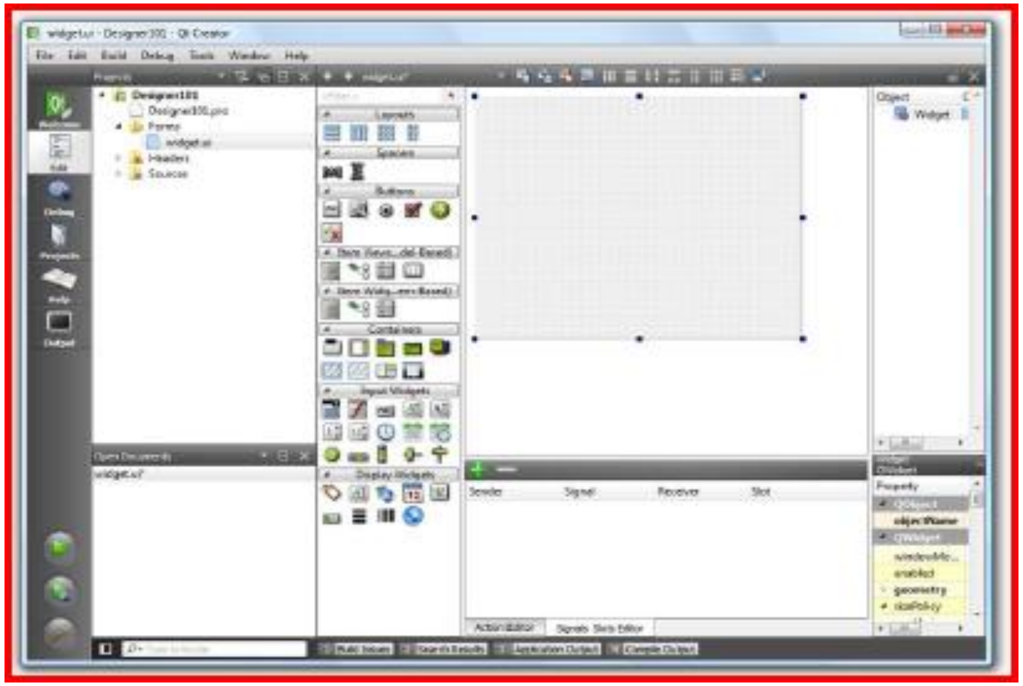
Sometimes it is necessary to show a message to confirm some action

```
import sys
from PyQt4 import QtGui
class MessageBox(QtGui.QMainWindow):
 def __init__(self, parent=None):
QtGui.QMainWindow.__init__(self, parent)
self.setGeometry(300, 300, 250, 150)
self.setWindowTitle('message box')
self.button=QtGui.QPushButton('Close')
self.setCentralWidget(self.button)
self.button.clicked.connect(self.close)
 def closeEvent(self, event):
reply = QtGui.QMessageBox.question(self, 'Message',"Are you sure to quit?",
QtGui.QMessageBox.Yes,QtGui.QMessageBox.No)
if reply == QtGui.QMessageBox.Yes:
        event.accept()
else:
        event.ignore()
app = QtGui.QApplication(sys.argv)
qb = MessageBox()
qb.show()
sys.exit(app.exec_())
```



Building GUI using QtDesigner

Qt Designer is the Qt tool for designing and building graphical user interfaces. It allows you to design widgets, dialogs or complete main windows using on-screen forms and a simple drag-and-drop interface.
It has the ability to preview your designs to ensure they work as you intended.



Building GUI using QtDesigner

- Qt Designer uses XML .ui files to store designs and does not generate any code itself.
- PyQt4 includes the `uic` Python module.
- PyQt4's `pyuic4` utility is a command line interface to the uic module
- `pyuic4` takes a Qt4 user interface description file and compiles it to Python code.
- The Python code is structured as a single class that is derived from the Python object type.
- The name of the class is the name of the toplevel object set in Designer with Ui_ prepended.
- The class contains a method called setupUi().

This takes a single argument which is the widget in which the user interface is created.

Building GUI using QtDesigner

Demo on QtDesigner usage.

Basic working order:
1. Place widgets roughly
2. Apply layouts from the inside out, add spacers as needed
3. Make connections
4. Use from code

□

Throughout the process, alter and edit properties

Building GUI using QtDesigner

Demo on QtDesigner usage.
4 - Use from code

NOTE:
To generate `.py` code from .ui file:

– `pyuic4 filename.ui -o Ui_filename.py`

To generate `.py` code from `.qrc` resource file :

– `pyrcc4 -o resources_rc.py resources.qrc`

To use ui interaface:

```
from uiMainWindow import Ui_MainWindow
class MyMainWindow(QtGui.QMainWindow):
    def __init__(self,parent):
        QtGui.QMainWindow.__init__(self, parent=None)
        self.ui=Ui_MainWindow()
        self.ui.setupUi(self)
        self.ui.actionOpen.triggered.connect(self.OpenFile
```

Embedding Matplolib in Qt

Matplotlib + IPython is very handy for interactive plotting, experimenting with datasets,trying different visualization of the same data, and so on. There will be cases where we want an application to acquire, parse, and then, display our data.
We will present some examples of how to embed Matplotlib in applications that use Qt4 as the graphical interface library.

We will see:

- How to embed a Matplotlib Figure into a Qt window
- How to embed both, Matplotlib Figure and a navigation toolbar into a Qt window
- How we can use QtDesigner to design a GUI for Qt4 and then embed Matplotlib into it

Embedding Matplolib in Qt

Necessary step:
- Import the `Figure Matplotlib object`: this is the backend-independent representation of our plot.
- Import from the `matplotlib.backends.backend_qt4agg` the module `FigureCanvasQTAgg` class, which is the backend-dependent figure canvas; It contains the backend-specific knowledge to render the Figure we've drawn.
-Import from `matplotlib.backends.backend_qt4agg` `NavigationToolbar2QTAgg` class

NOTES:
Note that `FigureCanvasQTAgg`, other than being a Matplotlib class, is also a `QWidget`—the base class of all user interface objects.
So this means we can treat `FigureCanvasQTAgg` like a pure Qt widget Object. `NavigationToolbar2QTAgg` also inherits from QWidget, so it can be used as Qt objects in a QApplication.

Example

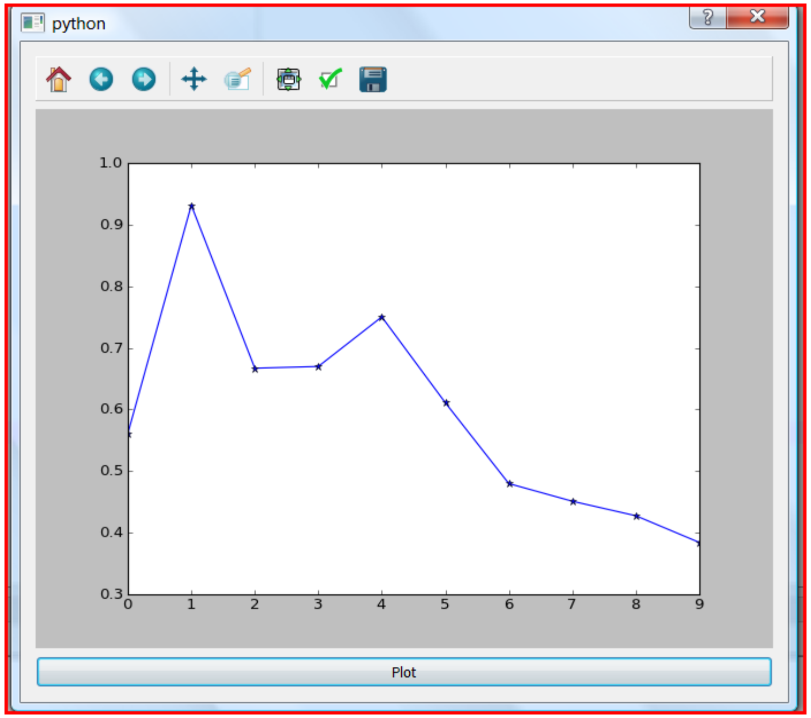
```
import sys
from PyQt4 import QtGui

from matplotlib.backends.backend_qt4agg import FigureCanvasQTAgg as
FigureCanvas
from matplotlib.backends.backend_qt4agg import NavigationToolbar2QTAgg as
NavigationToolbar
import matplotlib.pyplot as plt
import random

if __name__ == '__main__':
    app = QtGui.QApplication(sys.argv)

    main = Window()
    main.show()

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

Example

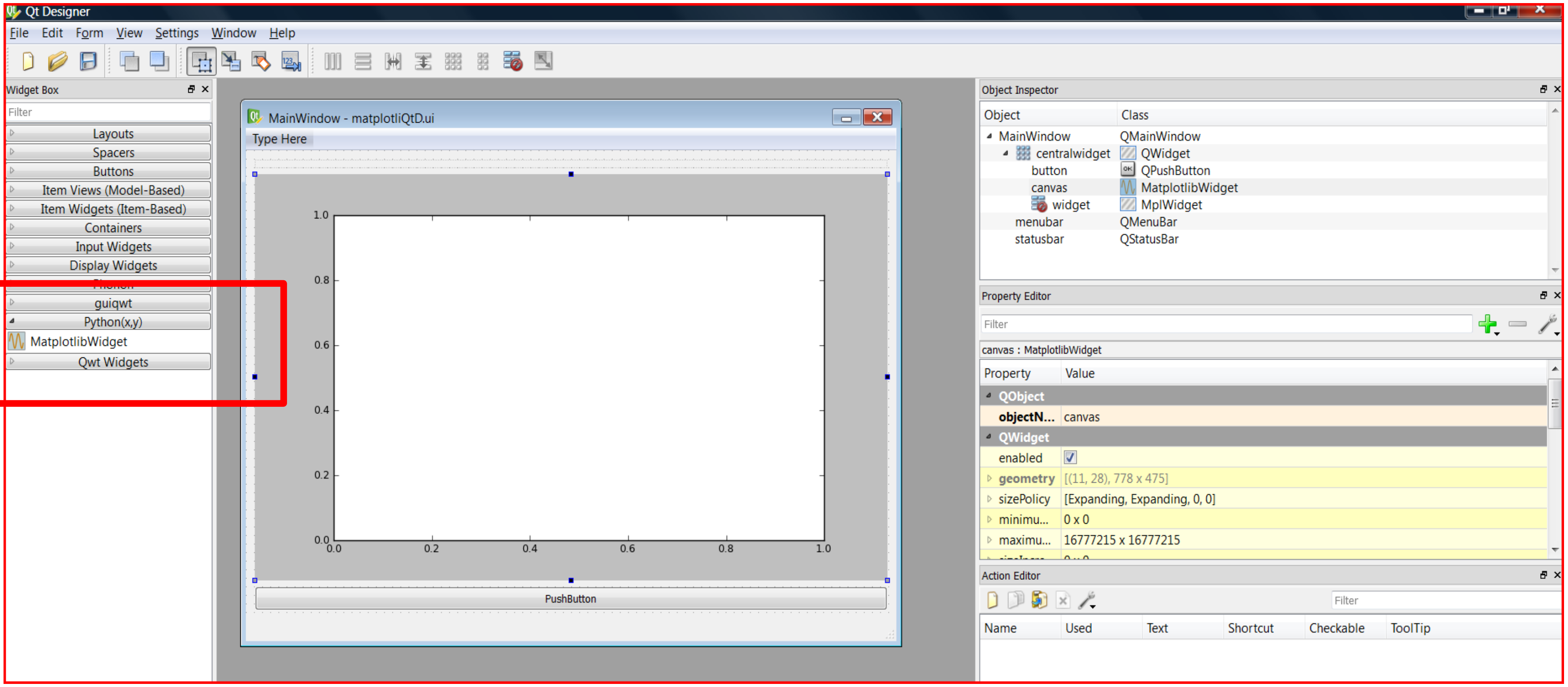
```
class Window(QtGui.QDialog):
    def __init__(self, parent=None):
        QtGui.QDialog__init__(parent)
        self.figure = plt.figure()
        # this is the Canvas Widget that displays the `figure`
        self.canvas = FigureCanvas(self.figure)
        # this is the Navigation widget
        self.toolbar = NavigationToolbar(self.canvas, self)
        # Just some button connected to `plot` method
        self.button = QtGui.QPushButton('Plot')
        self.button.clicked.connect(self.plot)
        # set the layout
        layout = QtGui.QVBoxLayout()
        layout.addWidget(self.toolbar)
        layout.addWidget(self.canvas)
        layout.addWidget(self.button)
        self.setLayout(layout)
```

Example

```
def plot(self):
        # random data
        data = [random.random() for i in range(10)]
        # create an axis
        ax = self.figure.add_subplot(111)
        # discards the old graph
        ax.hold(False)
        # plot data
        ax.plot(data, '*-')
        # refresh canvas
        self.canvas.draw()
```



Embedding Matplolib with QtDesigner



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
button	QPushButton
canvas	MatplotlibWidget
widget	MplWidget
menubar	QMenuBar
statusbar	QStatusBar

canvas : MatplotlibWidget

Property	Value
QObject	
objectN...	canvas
QWidget	
enabled	✓
geometry	[(11, 28), 778 x 475]
sizePolicy	[Expanding, Expanding, 0, 0]
minimu...	0 x 0
maximu...	16777215 x 16777215

Name	Used	Text	Shortcut	Checkable	ToolTip

Embedding VTK in Qt

VTK widget can be embedded as well in a PyQt Application. Once VTK is installed with python bindings, all you need to do is to let your GUI find these bindings, then import them into your GUI.

`QVTKRenderWindowInteractor` uses a `vtkGenericRenderWindowInteractor` to handle the interactions and it uses `GetRenderWindow()` to get the `vtkRenderWindow`.

Embedding VTK in Qt: Example

```
from PyQt4 import QtGui
from vtk.qt4.QVTKRenderWindowInteractor import QVTKRenderWindowInteractor
import vtk
import sys
from Ui_MainWindowVtk import Ui_MainWindow

if __name__=='__main__':
app = QtGui.QApplication(['QVTKRenderWindowInteractor'])
window=MyMainWindow()
window.show()
sys.exit(app.exec_())
```

Summer School on SCIENTIFIC VISUALIZATION

Embedding VTK in Qt: Example

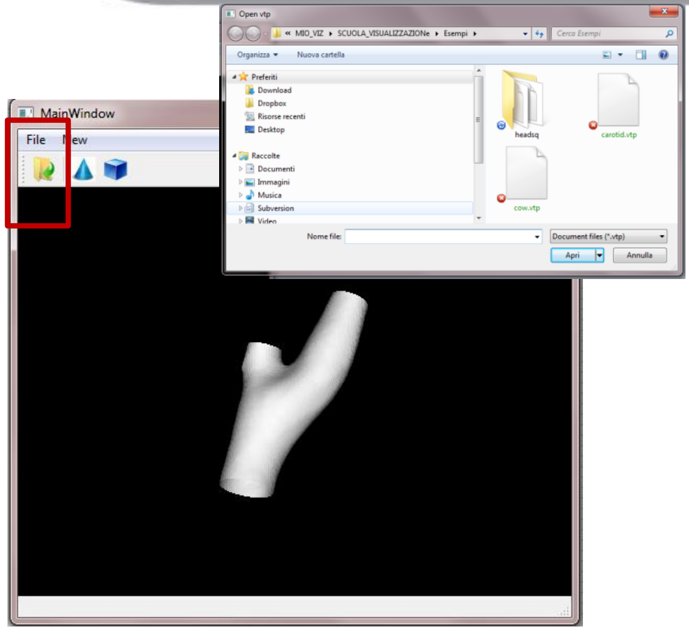
```
class MyMainWindow(QtGui.QMainWindow):
    def __init__(self,parent=None):
        QtGui.QMainWindow.__init__(self)
        self.ui=Ui_MainWindow()
        self.ui.setupUi(self)
        self.widget=QVTKRenderWindowInteractor(self)
        self.widget.Initialize()
        self.widget.Start()
        self.widget.show()
        self.setCentralWidget(self.widget)
        self.ui.actionCone.triggered.connect(self.addCone)
        self.ui.actionCube.triggered.connect(self.addCube)
        self.ui.actionOpen.triggered.connect(self.addFig)
```

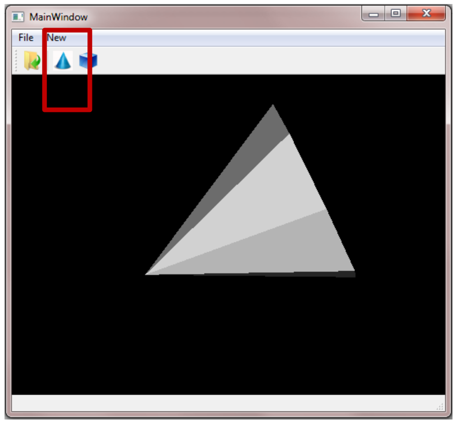
Embedding VTK in Qt: Example

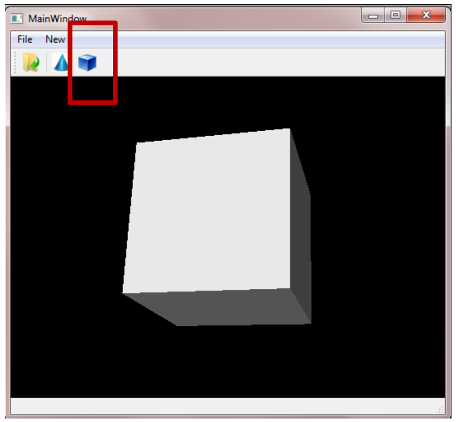
```
def addFig(self):
        filename=QtGui.QFileDialog.getOpenFileName(self,'Open vtp','.',"Document files (*.vtp);;All
            files (*.*)")
        reader = vtk.vtkXMLPolyDataReader()
        reader.SetFileName(str(filename))
        reader.Update()
        ren = vtk.vtkRenderer()
        self.widget.GetRenderWindow().AddRenderer(ren)
        mapper = vtk.vtkPolyDataMapper()
        mapper.SetInput(reader.GetOutput())
        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        ren.AddActor(actor)
def addCone(self):
        ren = vtk.vtkRenderer()
        self.widget.GetRenderWindow().AddRenderer(ren)
        cone = vtk.vtkConeSource()
        cone.SetResolution(8)
        coneMapper = vtk.vtkPolyDataMapper()
        coneMapper.SetInputConnection(cone.GetOutputPort())
        coneActor = vtk.vtkActor()
        coneActor.SetMapper(coneMapper)
        ren.AddActor(coneActor)
```

Embedding VTK in Qt: Example

```
def addCube(self):
        ren = vtk.vtkRenderer()
        self.widget.GetRenderWindow().AddRenderer(ren)
        cube = vtk.vtkCubeSource()
        cube = vtk.vtkCubeSource()
        cube.SetXLength(200)
        cube.SetYLength(200)
        cube.SetZLength(200)
        cube.Update()
        cubeMapper = vtk.vtkPolyDataMapper()
        cubeMapper.SetInputConnection(cube.GetOutputPort())
        cubeActor = vtk.vtkActor()
        cubeActor.SetMapper(cubeMapper)
        ren.AddActor(cubeActor)
        ren.ResetCamera()
```


MainWindow
File
New
Open vtp
MIO_VIZ ▸ SCUOLA_VISUALIZZAZIONe ▸ Esempi ▸
Cerca Esempi
Organizza
Nuova cartella
Preferiti
Download
Dropbox
Risorse recenti
Desktop
Raccolte
Documenti
Immagini
Musica
Subversion
Video
headsq
carotid.vtp
cow.vtp
Nome file:
Document files (*.vtp)
Apri
Annulla


MainWindow
File
New


MainWindow
File
New