HZB Helmholtz Zentrum Berlin

Graphical user interface design with Python & Qt

Karine Sparta, Macromolecular Crystallography Group, Helmholtz-Zentrum Berlin

Graphical user interface

Graphical User Interface (GUI)

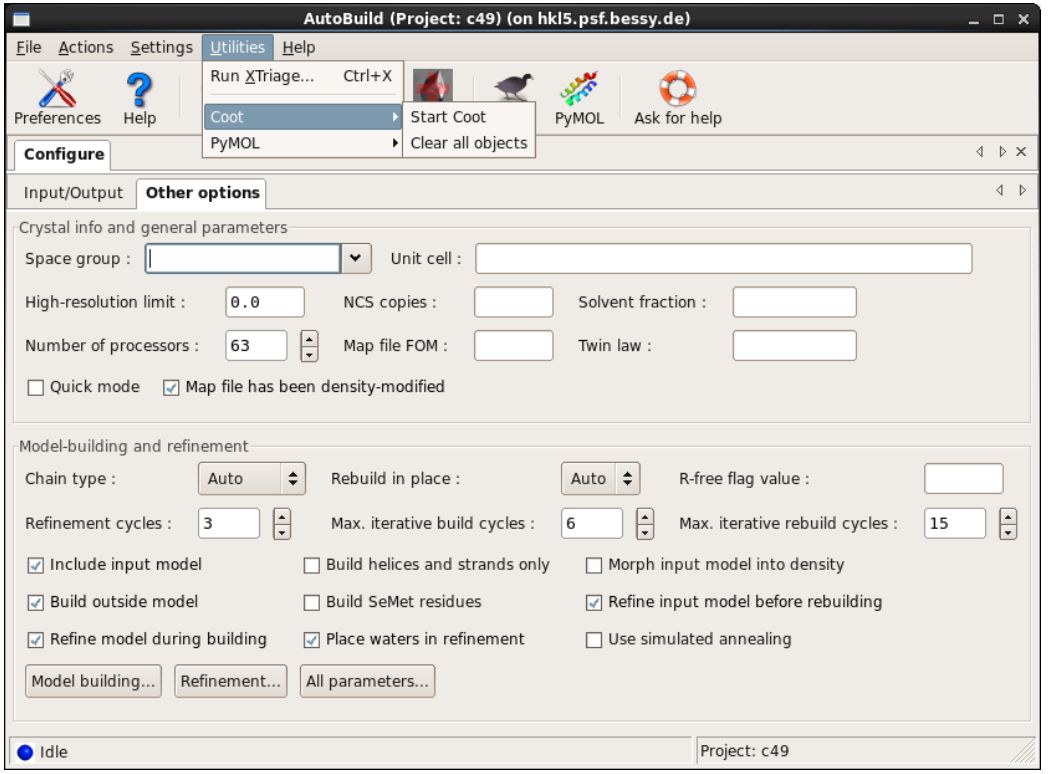
- Opposed to command-line interfaces (CLI)
- Intuitive interaction of the user with a device through widgets

GUIs are designed with the user in mind

- Aesthetic matters
- Clear and understandable
- Easy to work with

Elements of GUIs

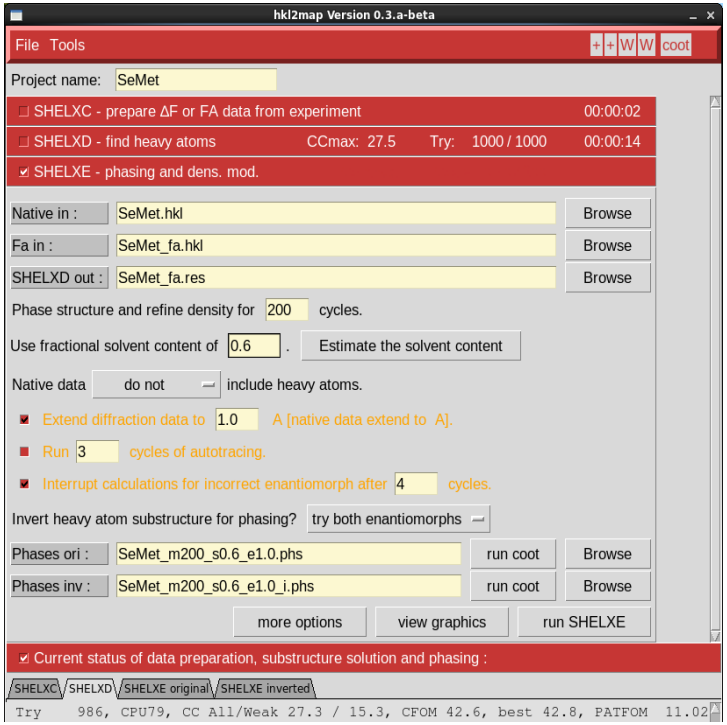
- Windows
- Menus
- Widgets
- Tabs



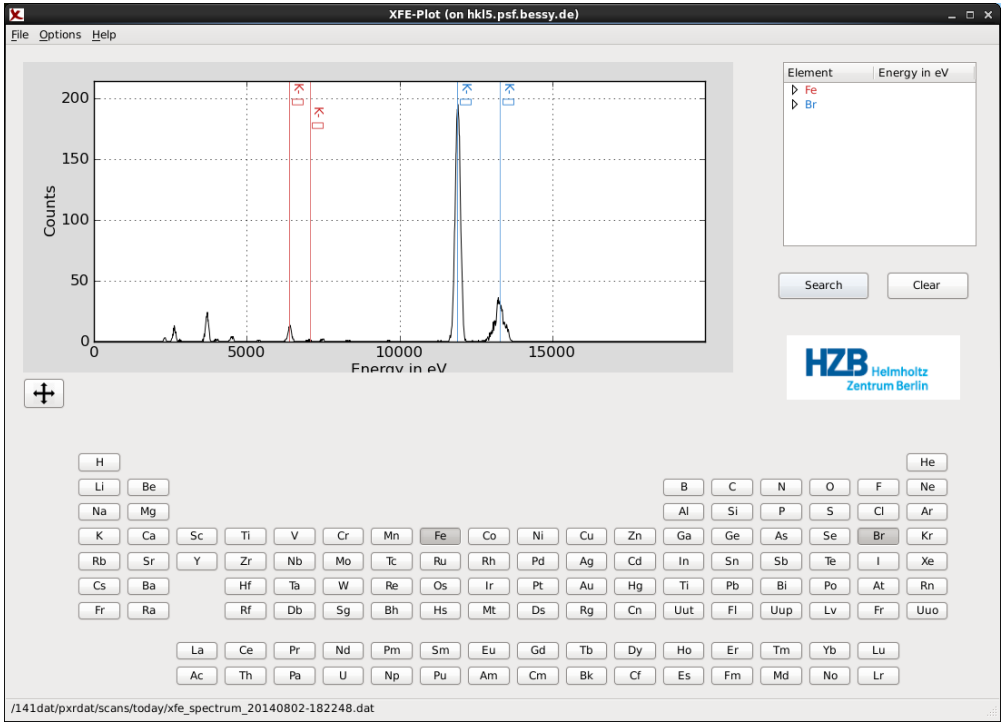
Widget toolkits for GUIs

Libraries for graphical control elements (widgets)

- OS specific (Cocoa for Mac OS X)
- Cross-platform (GTK, Qt, Adobe Flash)



hkl2map



xfeplot

Qt and Python bindings

HZB Helmholtz Zentrum Berlin

Qt is an application framework developed by the Qt Company and the Qt Project

- Initially written by Nokia for C++
- Several modules: `QtCore`, `QtGui`, `QtDesigner`...
- Cross-platform
- Uses native style APIs

Bindings for other programming languages

- Qt Jambi, PHP-Qt, QtRuby, qtcl…

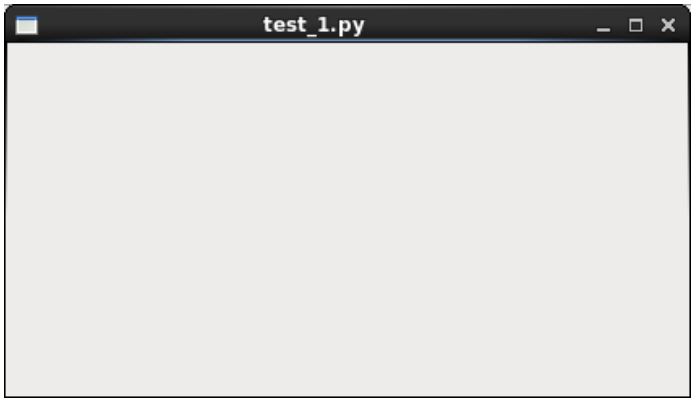
Python bindings

PyQt	PySide
https://www.riverbankcomputing.com/software/pyqt/intro	https://pypi.python.org/pypi/PySide/1.2.2
GNU GPL v3	LGPL
PyQt4, PyQt5	No Qt5.x support

First example

Create an empty window

```
#!/usr/bin/python
import sys
from PyQt4.QtGui import *

app = QApplication(sys.argv)

frame = QWidget()
frame.show()

app.exec_()
```



`sys.arg` lists the command line arguments passed to the Python script

The `QApplication` contains the *main event loop* (see slide 10)

`app.exec_()` starts the application

All user interface objects inherit the `QWidget` class

Built-in features: minimize, maximize, resize, move, close…

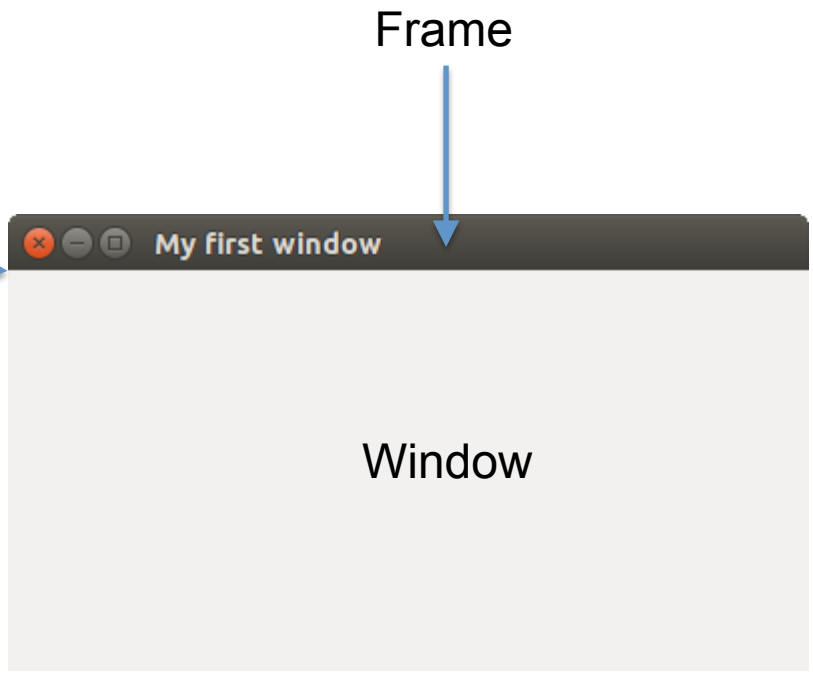
First example

Objects of the `QWidget` or any other class have *properties*

Use the *public functions* of the class to manipulate the properties of the object

http://doc.qt.io/qt-4.8/qwidget.html

```
#!/usr/bin/python
import sys
from PyQt4.QtGui import *

app = QApplication(sys.argv)

frame = QWidget()
frame.setWindowTitle("My first window")
frame.setGeometry(90, 50, 400, 200)
frame.show()

app.exec_()
```



A window with class

Same result as before, but now `frame` is an instance of the custom class `MyWindow`

```
#!/usr/bin/python
import sys
from PyQt4.QtGui import *

class MyWindow(QMainWindow):

    def __init__(self):
        super(MyWindow, self).__init__()
        self.setWindowTitle("My first window")
        self.setGeometry(90, 50, 400, 200)
        self.show()

app = QApplication(sys.argv)

frame = MyWindow()

app.exec_()
```

```
#!/usr/bin/python
import sys
from PyQt4.QtGui import *

class MyWindow(QMainWindow):

    def __init__(self):
        super(MyWindow, self).__init__()
        self.initMyWindow()

    def initMyWindow(self):
        self.setWindowTitle("My first window")
        self.setGeometry(90, 50, 400, 200)
        self.show()

app = QApplication(sys.argv)

frame = MyWindow()

app.exec_()
```

`QMainWindow` is a subclass of `QWidget` that also supports toolbars, statusbars, docking areas and central widgets

Buttons

```
#!/usr/bin/python
import sys
from PyQt4.QtGui import *

class MyWindow(QMainWindow):

    def __init__(self):
        super(MyWindow, self).__init__()
        self.myInit()

    def myInit(self):
        self.setWindowTitle("Button")
        self.setGeometry(90, 50, 400, 200)
        myButton = QPushButton('A cute button', self)
        myButton.setToolTip('Click me')
        myButton.move(150, 80)
        self.show()

app = QApplication(sys.argv)

frame = MyWindow()

app.exec_()
```

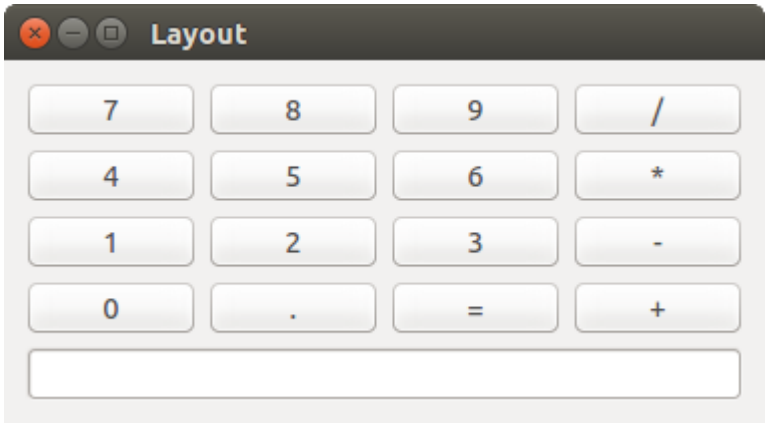
New class `QPushButton`

The tool tip is an attribute of many QWidgets

`setToolTip` sets the help text that appears when hovering with the mouse over the object

Layouts

The `QGridLayout` divides the window into columns and rows



Widgets may span several rows and/ or columns

`QLineEdit` is a one-line text editor

```
#!/usr/bin/python
import sys
from PyQt4.QtGui import *

class MyWindow(QWidget):

    def __init__(self):
        super(MyWindow, self).__init__()
        self.initMyWindow()

    def initMyWindow(self):
        myGrid = QGridLayout()
        self.setLayout(myGrid)

        blabels = [["7", "8", "9", "/"],
                   ["4", "5", "6", "*"],
                   ["1", "2", "3", "-"],
                   ["0", ".", "=", "+"]]
        for i in range(4):
            for j in range(4):
                button = QPushButton(blabels[i][j])
                myGrid.addWidget(button, i, j)

        myField = QLineEdit()
        myGrid.addWidget(myField, 4, 0, 1, 4)

        self.move(90, 50)
        self.setWindowTitle("Layout")
        self.show()

app = QApplication(sys.argv)
frame = MyWindow()
app.exec_()
```

Events

Widget toolkits are based on event-driven programming

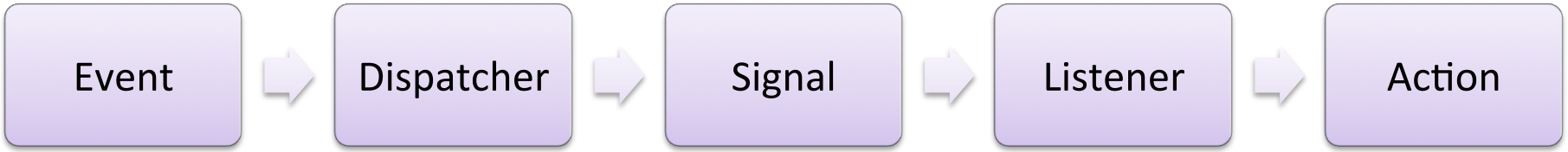
An event is anything that happens during the GUI execution

- User action: mouse click, keyboard input
- Messages from other processes



Event handling

- The system *listens* for specific events by running an *event loop*
- All widgets in a GUI notify events by emitting *signals*
- To handle an event, *connect* its signal to an action *slot*
- Signals that are not listened for are discarded
- If an event is detected, an action is triggered



Signals and slots

HZB Helmholtz Zentrum Berlin

In PyQt, events are handled using the `QtCore` module

- All other PyQt modules rely on QtCore
- QtCore contains non graphical libraries

Event handling involves three participants: the event source, the event object (signal) and the event target (slot). A signal must be connected to a slot to be handled. The slot can be any Python callable.

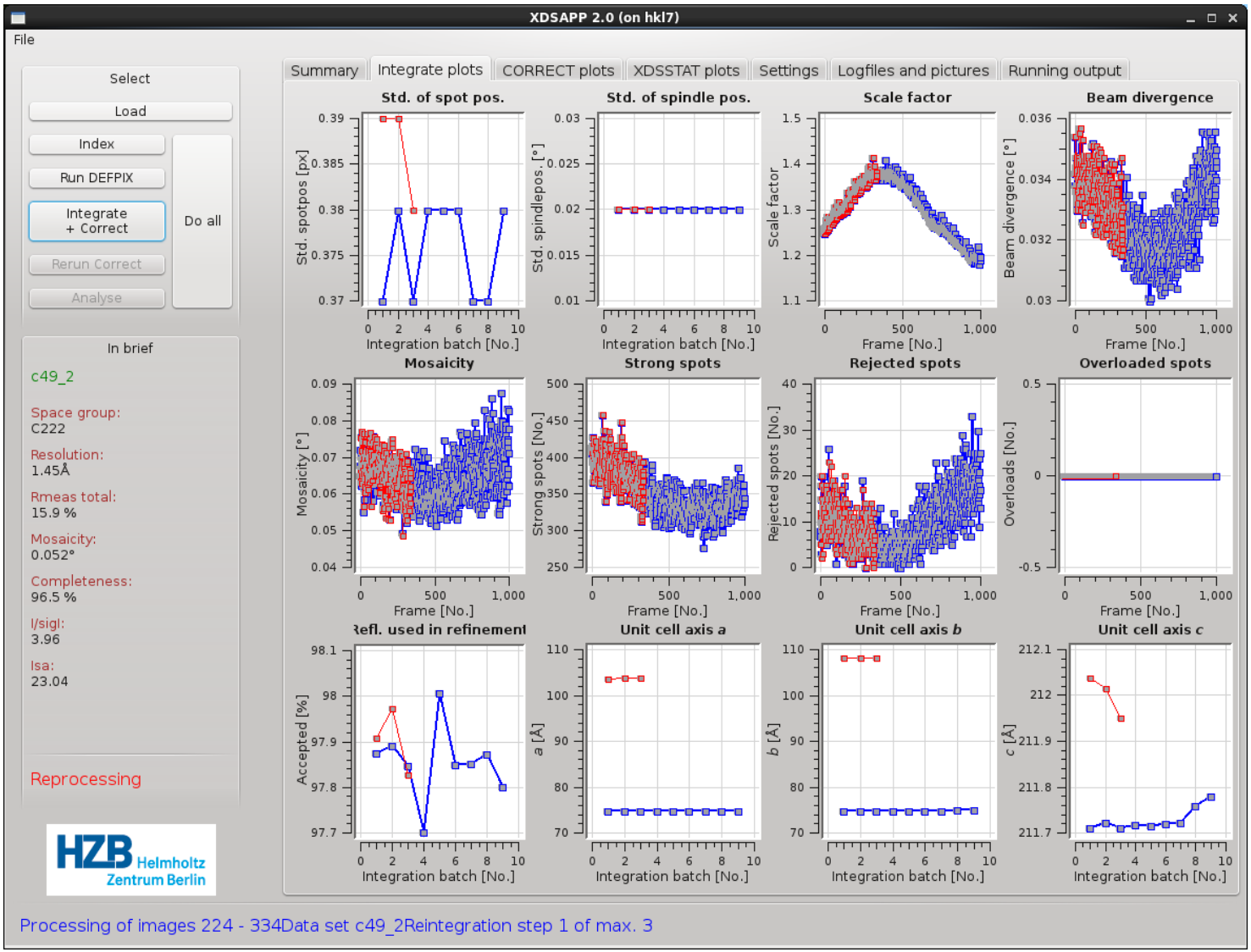
“Old style” connection between signals and slots

```
QtCore.QObject.connect(myButton, QtCore.SIGNAL('clicked()'), self.doThis)
```

“New style” connection between signals and slots (since PyQt4.5)

```
myButton.clicked.connect(self.doThis)
```

In practice: XDSAPP



Krug M., Weiss M. S., Mueller U., Heinemann U. (2012). *J. Appl. Cryst.* **45**, 568-572.

In practice: XDSAPP

HZB Helmholtz Zentrum Berlin

custom signals

```
self.connect(self.thread, SIGNAL("started()"), self.inactivate_buttons)
self.connect(self.thread, SIGNAL("finished()"), self.set_idle)
self.connect(self.thread, SIGNAL("finished()"), self.sig_finished)
self.connect(self.thread, SIGNAL("finished()"), self.sum_up)
self.connect(self.thread, SIGNAL("sum_up(QString)"), self.sum_up)
self.connect(self.thread, SIGNAL("terminated()"), self.terminated)
self.connect(self.thread, SIGNAL("output(QString)"), self.updateGui)
self.connect(self.thread, SIGNAL("update_brief(QString)"), self.update_brief)
self.connect(self.thread, SIGNAL("update_operating_status(QString)"), self.update_operating_status)
self.connect(self.thread, SIGNAL("error(QString)"), self.errorqstring)
self.connect(self.thread, SIGNAL("replot_int(QString)"), self.doreplot_int)
self.connect(self.thread, SIGNAL("replot_correct(QString)"), self.doreplot_correct)
self.connect(self.thread, SIGNAL("replot_xdsstat(QString)"), self.doreplot_xdsstat)
self.connect(self.thread, SIGNAL("Status_Line(QString)"), self.updateStatusLine)
self.connect(self.thread, SIGNAL("Summary_Text(QString)"), self.updateSummaryText)
```

```
self.connect(self.Load, SIGNAL("pressed()"), self.select)
```

menu item signals

```
self.connect(self.actionSavePlots, SIGNAL("triggered()"), self.save_image)
self.connect(self.actionLoad, SIGNAL("triggered()"), self.select)
self.connect(self.actionSavesettings, SIGNAL("triggered()"), self.save_xdssettings)
self.connect(self.actionQuit, SIGNAL("triggered()"), self.close)
self.connect(self.select_logfile, SIGNAL("currentIndexChanged(int)"), self.get_logfile_list)
self.connect(self.actionAbout, SIGNAL("triggered()"), self.about)
self.connect(self.reload_logfile, SIGNAL("pressed()"), self.reload_logfile_list)
self.connect(self.load_xds_logpic, SIGNAL("pressed()"), self.open_xds_logpic)
```

button signals

```
self.connect(self.load_xdsstat_logpic, SIGNAL("pressed()"), self.open_xdsstat_logpic)
self.connect(self.Index, SIGNAL("pressed()"), self.doindex)
self.Index.setEnabled(False)
self.connect(self.DEFPIX, SIGNAL("pressed()"), self.run_defpix)
self.DEFPIX.setEnabled(False)
self.connect(self.Process, SIGNAL("pressed()"), self.doprocessing)
self.Process.setEnabled(False)
self.connect(self.rerun_correct, SIGNAL("pressed()"), self.doreruncorrect)
self.rerun_correct.setEnabled(False)
self.connect(self.button_analyse, SIGNAL("pressed()"), self.analyse)
self.button_analyse.setEnabled(False)
```

“Old style” syntax

Close button

A signal can be connected to many slots

A slot can be connected to many signals

What happens when you additionally connect the callable `self.show` to the `closeButton.clicked` signal?

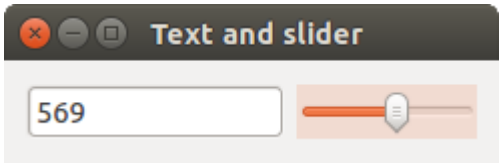
```
#!/usr/bin/python
import sys
from PyQt4.QtCore import *
from PyQt4.QtGui import *

class MyWindow(QMainWindow):

    def __init__(self):
        super(MyWindow, self).__init__()
        self.initMyWindow()

    def initMyWindow(self):
        self.setGeometry(90, 50, 200, 200)
        closeButton = QPushButton("Close", self)
        closeButton.move(50, 80)
        closeButton.clicked.connect(self.doThis)
        closeButton.clicked.connect(self.close)
        self.show()

    def doThis(self):
        print("The close button has been clicked")

app = QApplication(sys.argv)
frame = MyWindow()
app.exec_()
```

Interaction between widgets

The signals `valueChanged` and `textChanged` also send the values of the changed parameters

```
mySlider.valueChanged.connect
(myField.setText)
```

fails with a TypeError! Therefore the custom callables




```
class MyWindow(QWidget):

    def __init__(self):
        super(MyWindow, self).__init__()
        self.initMyWindow()

    def initMyWindow(self):
        global myField, mySlider
        myField = QLineEdit()
        mySlider = QSlider(Qt.Horizontal)
        mySlider.setRange(0, 1000)

        hLayout = QHBoxLayout()
        self.setLayout(hLayout)
        hLayout.addWidget(myField)
        hLayout.addWidget(mySlider)

        mySlider.valueChanged.connect(self.updateField)
        myField.textChanged.connect(self.updateSlider)

        self.setWindowTitle("Text and slider")
        self.show()

    def updateField(self, value):
        myField.setText(str(value))

    def updateSlider(self, text):
        try:
            mySlider.setValue(int(text))
        except ValueError:
            pass

app = QApplication(sys.argv)
frame = MyWindow()
app.exec_()
```

Simple calculator

Create one using what you have learned!

Tip: use the `eval(text)` function