MUEI/MUNR

ETSEIB

Course on Medical Imaging
2023-2024

Session 4 - Lab
User interface

Dani Tost

PyQt

So far, we've tested the code in the terminal. This is slow. We need an interface to input parameters and allow us to do several things in any order in run time.

Interfaces are needed in all kind of applications

For that, we will learn PyQt5, a library to design interfaces.

PyQt

https://www.qt.io/

- Library for the creation of interfaces
- Cross-platform (including android and iOS)
- Tutorials, manual pages: http://qt-project.org/
- Compatible with different programming languages.
- Binding with python: https://wiki.python.org/moin/PyQt
- The code can be partially created graphically with qtDesigner (we won't)
- We will use PyQt5 https://pypi.org/project/PyQt5/

Main loop and events

A GUI application is **event-driven: it** manages all interactions and tasks

GUI applications consist of an loop in which at each iteration all events succeeded between the previous iteration and the current one(that have been queued) are processed until an event makes the loop finish.

Events are generated mainly:
- by the user
- a timer
- the window manager etc..

Basic instructions:

```
app = Qapplication(sys.argv)
```

Creates an application

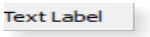
```
app.exec()
```

starts the main loop

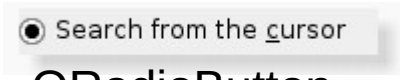
http://doc.qt.io/qt-5/qapplication.html

QtWidgets

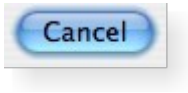
- Widgets are the basic elements of user interfaces in Qt.
- Widget display information
- Widgets receive user input
- Widgets can be containers of other widgets
- Widgets can be parameterized and customized
- There are prototyped widgets: buttons, checkboxes...



QLabel



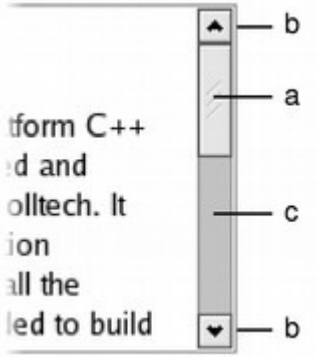
QRadioButton



QPushButton

QToolbox

QCheckBox

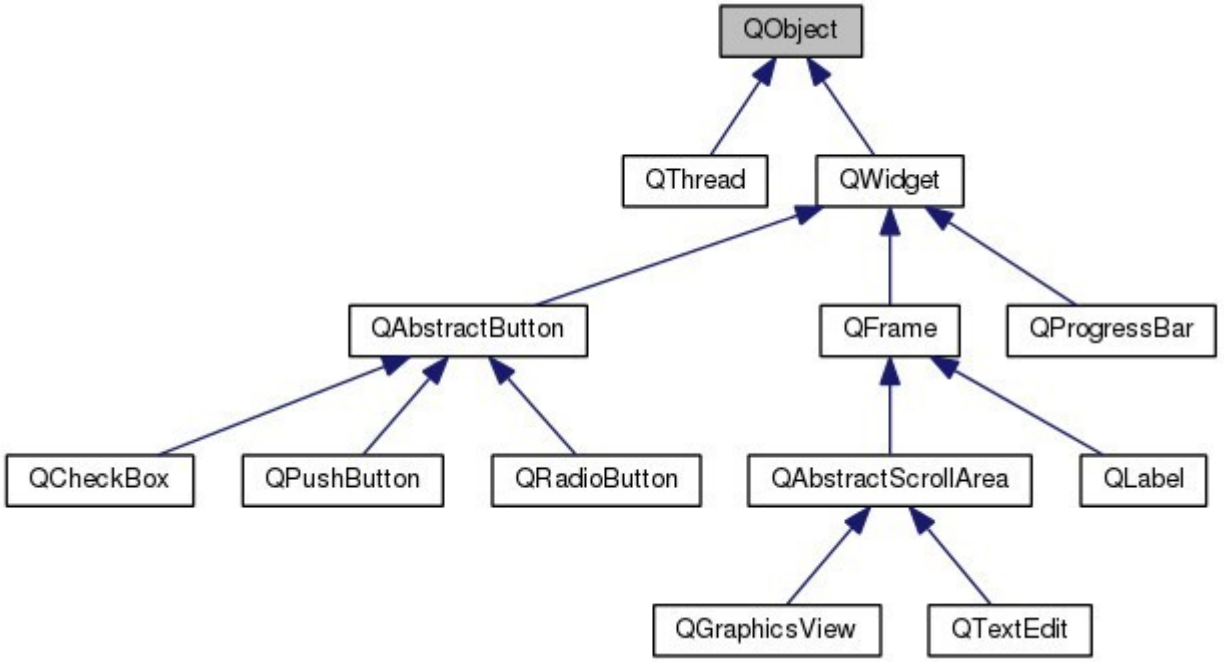


QScrollBar

(All images from http://doc.qt.io/qt-5/)

QtWidgets

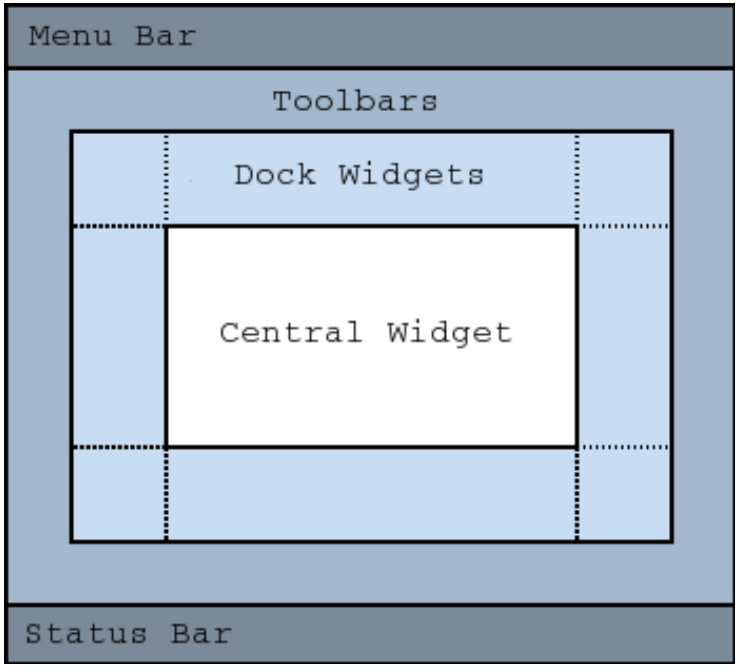
QtWidgets are based on the inheritance mechanism



From https://wiki.qt.io/Qt_for_Beginners

QtWindows

- **Windows** are widgets no embedded in other widgets
- **QMainWindow** is the widget used to create the main interface of the application. It has a predefined layout.
- **QtLayouts** provide means of organizing widgets in Window
- **QDialog** windows are secondary windows on which users choose options and values.
- There are standard **dialog windows**
  - ColorDialog
  - FileDialog
  - FontDialog
  - MessageDialog
  - And you can create your own



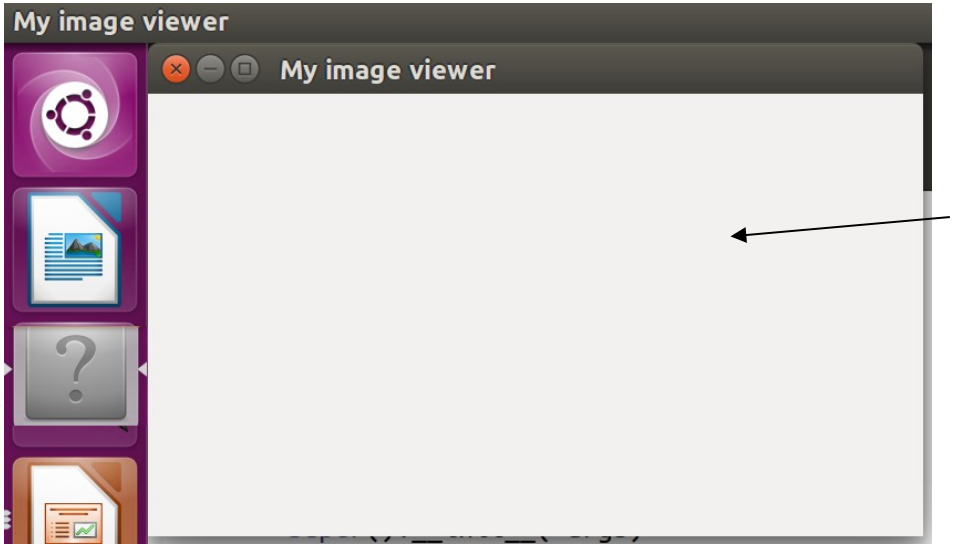
Practice

Download the examples and deploy the code in a separate directory (do not remove your own code). In the directory examples, run interface_ex1.py.

```
$ python3 interface-ex1.py
```

The name of the active application is written on the ubuntu desktop menubar

The icon of the application



An independent window called My image viewer has been created.

```
if __name__=='__main__':
    app = QApplication(sys.argv)
    p = MainWindow()
    sys.exit(app.exec_())
```

Look at the main program. A QApplication (app) is created. Within the context of this QApplication a main window widget is created. The QApplication runs until it finishes, because an event forces it to and the program is closed.

Practice

Look at the class `MainWindow`. It inherits from the class `QMainWindow` and thus has its attributes and methods.

```
class MainWindow(QMainWindow):
    def __init__(self, *args):
        super().__init__(*args)
        self.initUI()

    def initUI(self):
        self.setWindowTitle("My image viewer")
        self.setGeometry(0, 0, 700, 400)
        self.show()
```

This method is invoked whan an instance to the class is created. Self represents the created instance (equivalent to this in c++)

This method creates the interface

Try other sizes of the window. Try to change the title of the window.

Events handling

User interaction yields **events** that can be managed by the application

Slots are functions and methods that are called when a particular event occurs. Events emit a **signal** that connects to one or more specific slots or send another signal.

Widgets have predefined methods that are slots connected to signals emitted when specific events on the widget succeed: `closeEvent, dragEnterEvent, dragLeaveEvent, DragMoveEvent, focusInEvent, focusOutEvent, ketPressEvent, keyReleaseEvent ….`

We need to implement these methods to have the desire behavior.

Try to add to your widget the following method:

```
def closeEvent(self, event):
    print("Closing")
```

```
s$ python3 interface_ex1.py
Closing ...
```

By default the closeEvent launches the `close()` method. Let's add a confirmation widget . For that, use interface_ex2.py

Practice

Run interface_ex2.py. Look at the class MainWindow. It inherits from the class QMainWindow and, thus, it has its attributes and methods.

```
class MainWindow(QMainWindow):
    def __init__(self, *args):
        super().__init__(*args)
        self.initUI()

    def initUI(self):
        self.setWindowTitle("My image viewer")
        self.setGeometry(0, 0, 700, 400)
        self.show()

    def closeEvent(self, event):
        reply = QMessageBox.question(self, 'Message',
                                     "Are you sure to exit?", QMessageBox.Yes |
                                     QMessageBox.No, QMessageBox.No)

        if reply == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()
```

This method is invoked when an instance to the class is created.

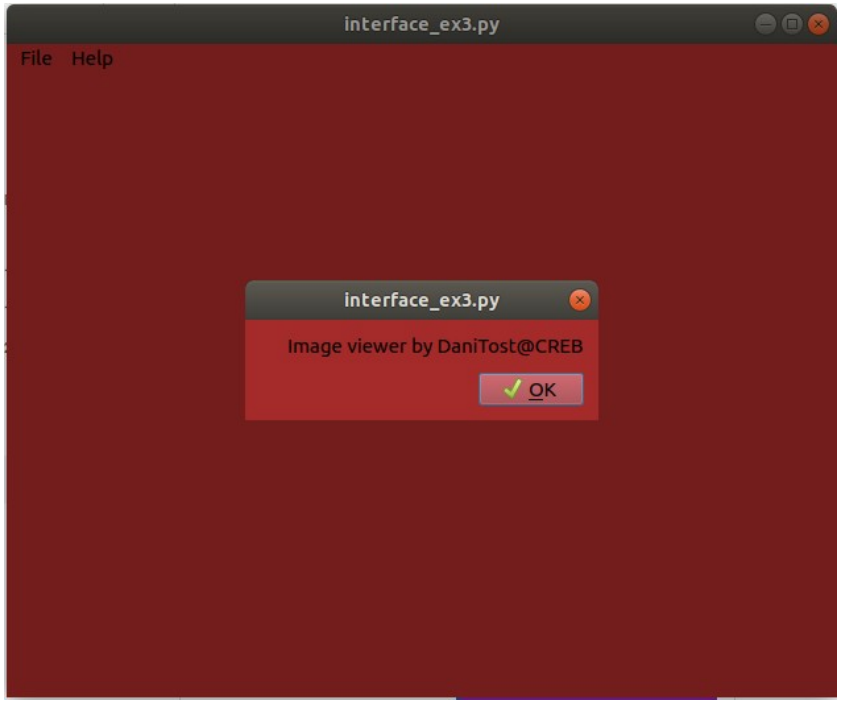
This method creates the interface

When we click on the cross we generate a "close" event that automatically generates a call to the method closeEvent. This method opens a QMessageBox allowing users to confirm if they want to exit or not.

Menus

QMainWindows have a main menu bar. You can add options to this menu that, in turn, can be submenus. For each option of the menu, you need to define which method will be launched and implement this option. Options can have an icon and a shortcut.

Download `interface_ex3.py` and run it. Observe that it has an embedded menu bar with two submenus. Try the different options. Time to analyze the code.



Menus

```
class MainWindow(QMainWindow):
    def __init__(self, *args):
        super().__init__(*args)
        self.initUI()

    def initUI(self):
        self.setStyleSheet("background-color: brown;")
        self.create.menus()
        self.show()

    def create_menus(self) :
        self.menubar = self.menuBar()
        self.create_file_menu()
        self.create_help_menu()

    def create_file_menu(self):
        self.menuFile = self.menubar.addMenu('File')
        exitAction = QAction(QIcon('exit.png'), 'Exit', self)
        exitAction.setShortcut('Ctrl+Q')
        exitAction.setStatusTip('Exit application')
        exitAction.triggered.connect(self.close )
        self.menuFile.addAction(exitAction)
```

The embedded menu bar

The file menu bar

We define the exit action

We add this action to the File menu

The `close` method will be launched when users will click on the option ‘Exit’ and generate an event of closing the application. This event will yield invoking the `closeEvent` method.

Menus

```
def closeEvent(self, event):
    reply = QMessageBox.question(self, 'Message',
                                 "Are you sure to exit?", QMessageBox.Yes |
                                 QMessageBox.No, QMessageBox.No)

    if reply == QMessageBox.Yes:
        event.accept()
    else:
        event.ignore()
```

The `closeEvent` methods shows a `QMessageBox.question` dialog widget. If the user chooses Yes the event is accepted and thus the application is closed.

The `closeEvent` is a special method of PyQt related with the event of closing the window.

Look at the Help menú and the `help_action` code. This action is not related to any event. It is launched when users press the option help. It opens a diffrent type of dialog: `QMessageBox`. There are many more types of dialogs. Check in the reference manual

```
def help_action(self) :
    text = 'Option in progress sorry'
    msgBox = QMessageBox(self)
    msgBox.setText(text)
    msgBox.exec_()
```

PyQt

So far, we've tested the code in the terminal. This is slow. We need an interface to input parameters and allow us to do several things in any order in run time.

Interfaces are needed in all kind of applications

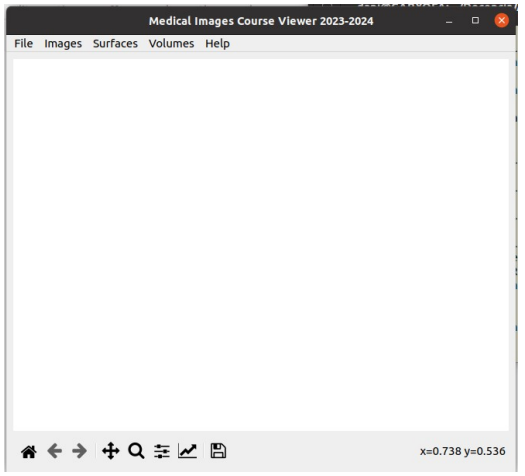
For that, we will learn PyQt5, a library to design interfaces.

Go to the common git project and pull:

```
git pull
```

Try to run the program:

```
python3 viewer.py
```



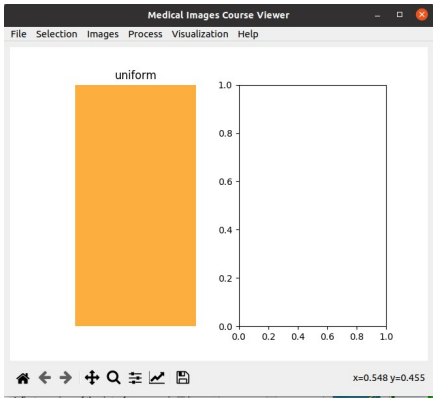
A first version of the interface

Make a pull to update the project **medical_images_2023_ 2024:**

```
git pull
```

Try to run the program:

```
python3 viewer.py
```



We'll try to understand it and add other options.

A first version of the interface

In addition to the files you already know, it contains the following files:

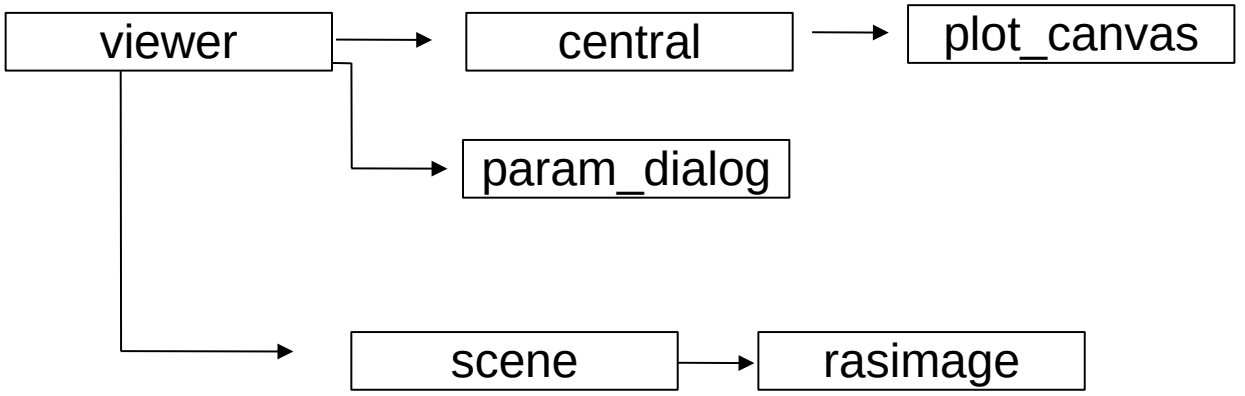
- scene.py: a container of images (and later of 3D objects)

- viewer.py: the main program that defines the menus and actions

- central.py: the central widget that contains a matplotlib window

- plot_canvas: the matplotlib window where the images are rendered

- param_dialog: the definition of the different dialog to get parameters for each option of the menu.



A first version of the interface

Analyze the code. The main window has an attribute: `scene`, that is a container of objects (images, surfaces and volumes)

In `initUI` the menus are created, then a central widget is created (implemented in the module `central.py`) which creates the matplotlib window.

```
class MainWindow(QMainWindow):

    def __init__(self, *args):
        super().__init__(*args)
        self.scene = Scene()
        self.initUI()
        self.init_error_dict()


    def initUI(self):
        self.create_menus()
        self.central = CentralWidget(self.scene)
        self.setCentralWidget(self.central)

        self.show()
```

Central widget

```
class CentralWidget(QWidget):
    def __init__(self, *args):
        """
        Returns a CentralWidget instance
        """
        super().__init__(*args)
        self.initUI()

    def initUI(self):
        """
        Initializes the layout. It creates a standard PlotCanvas widget to render images and
        a VTKWidget to render surfaces and volumes. Dependeing on what must be rendered either one or
        the other will be shown
        """

        self.layout = QVBoxLayout()
        self.visuwidget = PlotCanvas(self)
        self.layout.addWidget(self.visuwidget)
        self.layout.addWidget(Navbar(self.visuwidget, self))
        self.vtkwidget = VTKWidget(self)
        self.layout.addWidget(self.vtkwidget)
        self.vtkwidget.setHidden(True)
        self.setLayout(self.layout)
```

The Central widget has a Vertical (QHBox) layout. Each widget added to this central widget will be stacked below the previous one.

Currently, it has only two widgets: visuwidget an instance of the PlotCanvas class and vtkwidget, prepared for the 3D part.

Menus of options
Read

Observe the read file option. It will work as soon as you implement the corresponding class method in RasImage

```
def create_file_menu(self):
    self.menuFile = self.menubar.addMenu('File')
    readImageAction = QAction(QIcon('resources/icons/read.png'),  'Read image', self)
    readImageAction.triggered.connect(self.read_image)
    self.menuFile.addAction(readImageAction)
```

```
def read_image(self):
    """
    Fully implemented
    """
    options = QFileDialog.Options()
    filename, ok = QFileDialog.getOpenFileName(
        self, "Read image file", "", "", options=options
    )
    if ok:
        im = RasImage.read(filename)
        if im:
            name = self.scene.add_item(im)
            self.add_item_interface(name, 'RasImage')
        else:
            self.show_error(2)
```

plot_canvas class

```
from PyQt5.QtWidgets import QSizePolicy
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import matplotlib.pyplot as plt
from matplotlib.widgets import Cursor


class PlotCanvas(FigureCanvas):
    def __init__(self, parent=None):
        self.fig = Figure()
        super().__init__(self.fig)
        self.setParent(parent)
        self.sp = self.fig.add_subplot(1, 1, 1)
        self.sp.axis("off")
        self.nplots = 1

    def render_image(self, ima, pvisu):
        show_histo =  'show_histogram' in pvisu and pvisu['show_histogram']
        show_cdf =  'show_cdf' in pvisu and pvisu['show_cdf']
        if (not show_histo and not show_cdf) or ima.histogram is None:
            self.sp = self.fig.add_subplot(1, 1, 1)
            self.nplots = 1
            ima.draw_matplotlib(self.sp, pvisu)
        else:
            self.sp = [self.fig.add_subplot(1, 2, 1), self.fig.add_subplot(1, 2, 2)]
            ima.draw_matplotlib(self.sp[0], pvisu)
            ima.draw_histogram_matplotlib(self.sp[1])  # to be refined if we want to be able to select a
channel
            if show_histo and show_cdf:
                self.sp.append(self.sp[1].twinx())
                self.nplots = 3
                ima.draw_cdf_matplotlib(self.sp[1], pvisu)
            else:
                self.nplots = 2
        self.draw()
```

When an image is rendered 1 or 2 plots are defined depending on if the histogram and/or cdf must be rendered.

To render a histogram, the attribute histogram must exist. (idem cdf)

A first version of the interface

In your project.

1. Add an issue “Add the interface”.
2. Create the merge request
3. In the terminal, go to your project, fetch the new branch and checkout to it
4. Copy the interface files (not rasimage.py!!) to this repo
5. git add them, commit, push
6. Merge.

Try to add the procedural menu

In RasImage define a method that returns the list of names of the procedures you support.

In viewer:

```
self.menuProcedural = self.menuImage.addMenu("Procedural")
self.menuProcedural.setIcon(QIcon("resources/icons/proced.png"))
for name in RasImage.procedural_methods():
    proceduralAction = QAction(name, self)
    proceduralAction.triggered.connect(self.procedural_image)
    self.menuProcedural.addAction(proceduralAction)
```

Observe how it works!

Continue, complete the menú with what we've done so far.