PyQT GUIs

Python GUIs

- Python wasn't originally desined for GUI programming
- In the interest of "including batteries" the `tkinter` was included in the Python standard library
  - `tkinter` is a Python wrapper around the Tcl/Tk GUI toolkit
  - Tk must be installed on your system (included in most Unixes, additional install on Mac and Windows)
  - Tk is old and weak
- Many other GUI libraries were created for Python. wxPython, PyGTK, and PyQt/PySide the most popular
- PyQt/PySide was once difficult to install because Qt was difficult to install, but the Anaconda folks fixed that.
- So we'll use PyQt, PyQt5 to be precise. Install with:

```
$ conda install pyqt
```

PyQt

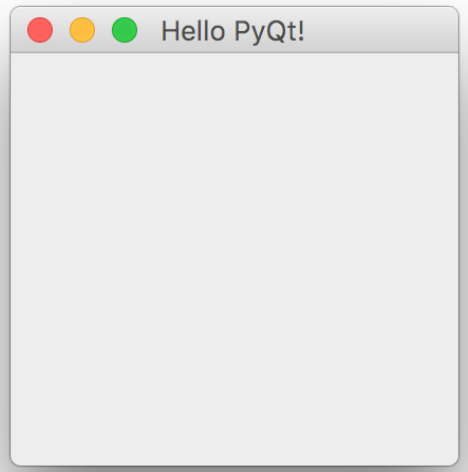
- Qt is a C++ library originally created by Norwegian company Troll Tech.
- Qt has always enjoyed a reputation as a well-designed and powerful GUI framework.
- The KDE project chose to base their popular KDE (K Desktop Environment) graphical shell for Linux.
- Like most modern GUI frameworks, Qt (and PyQt) makes heavy use of objects.

Hello, PyQt

Running helloqt.py

```
import sys
from PyQt5.QtWidgets import
      QApplication, QWidget

app = QApplication(sys.argv)

w = QWidget()
w.setWindowTitle('Hello PyQt!')
w.show()
return_code = app.exec_()
sys.exit(return_code)
```

will show:



Basic PyQt App Outline

1. Create a `QApplication` object
2. Create a main application window (`QWidget` object)
3. Set paramters of the main window, create and add child widgets, etc.
4. Show main application window
5. Start the app (`app.exec_()`)

Basic Qt Application Elements

Import the widgets we'll use:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
```

Create a `QApplication` object, passing the command-line arguments to the constructor:

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

Create the main window and set its paramters:

```
w = QWidget()
w.setWindowTitle('Hello PyQt!')
```

Show the main window and start the application:

```
w.show()
return_code = app.exec_()
sys.exit(return_code)
```

Structure of (Py)Qt Library

Three main PyQt modules:

- QtCore – Core non-GUI utilities
- QtGui – window system integration, event handling, 2D graphics and imaging, fonts
- QtWidgets – the basic elements of UIs: labels, buttons, text input, lists, tables, menus, toolbars

Also several special-purpose modules:

- QtMultimedia, QtBluetooth, QtNetwork, QtPositioning, Enginio, QtWebSockets, QtWebKit, QtWebKitWidgets, QtXml, QtSvg, QtSql, QtTest

Adding Child Widgets

In label.py we begin as before:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QLabel

app = QApplication(sys.argv)
w = QWidget()
w.setWindowTitle('Hello PyQt!')
```

A `QWidget` with no parent is a window. Note that *parent* means the owner of the widget on the screen, not the superclass.

We supply two Arguments to the `QLabel` constructor: the text (or image) to display, and the parent. Here the parent widget is our main application window, `w`:

```
lbl = QLabel('Hello, label!', w)
w.show()
sys.exit(app.exec_())
```

The label widget will appear in the upper left corner of it's parent widget, the main window. In future examples we'll see how to lay out widgets on their parents.

Buttons, Signals and Slots

In button.py the following creates a QPushButton with "Push it . . . " as its text and the main application window as its parent:

```
#!/usr/bin/env python3

import sys
from PyQt5.QtWidgets import QApplication, QWidget, QPushButton
app = QApplication(sys.argv)
w = QWidget()
w.setWindowTitle('Hello PyQt!')
btn = QPushButton('Push it ...', w)
```

QPushButton widgets emit *signals* when certain events happen, such as when they are clicked. These signals can be connected to any number of *slots*, which are simply Python callables (functions, methods, or lambda expressions). Here we connect the button to two slots, a defined function and a lambda expression:

```
def up():
    print("up!")

btn.clicked.connect(up)
btn.clicked.connect(lambda: print('real good!'))
w.show()
sys.exit(app.exec_())
```

Message Boxes

Four standard message boxes, invoked with one of these class methods:

- `QMessageBox.critical`
- `QMessageBox.information`
- `QMessageBox.question`
- `QMessageBox.warning`

Arguments are: parent, message box title, message, buttons, and (optional) default button.

For example, here's a simple information message box:

```
reply = QMessageBox.information(main_window,
                                "Message Box Title",
                                 "The message.",
                                 QMessageBox.Ok)
```

Here's a question:

```
reply = QMessageBox.question(self,
                             "Question",
                             "CS 2316 is the best, right?!",
                             QMessageBox.Yes | QMessageBox.No,
                             QMessageBox.Yes)
answer = "Yes" if reply == QMessageBox.Yes else "No"
```

Events

Some widgets emit events which can be handled. For example, when closing a window:

```
class MainWindow(QMainWindow):

    def __init__(self):
        super(MainWindow, self).__init__()

        btn = QPushButton('Quit', self)
        btn.clicked.connect(QCoreApplication.instance().quit)
        btn.resize(qbtn.sizeHint())
        btn.move(50, 50)
        left, top, width, height = 100, 100, 300, 200
        self.setGeometry(left, top, width, height)
        self.setWindowTitle('Quit button')

    def closeEvent(self, event):
        reply = QMessageBox.question(self,
                                     "Confirm",
                                     "Are you sure you want to quit?"
                                     QMessageBox.Yes | QMessageBox.No,
                                     QMessageBox.No)
        if reply == QMessageBox.Yes:
            event.accept()
        else:
            event.ignore()
```

See quitter.py.

Absolute Positioning

Previous examples used absolute positioning, for example:

```
self.setWindowTitle("Message Box Demo")
left, top, width, height = 50, 50, 300, 200
self.setGeometry(left, top, width, height)

info_btn = QPushButton("Information ...", self)
info_btn.move(50, 50)
info_btn.resize(info_btn.sizeHint())
```

This is brittle. For example, run message$_{\text{box}}$.py and resize the window to hide the buttons.

Layout Management

Here we use a QVBoxLayout layout manager to stack widgets vertically:

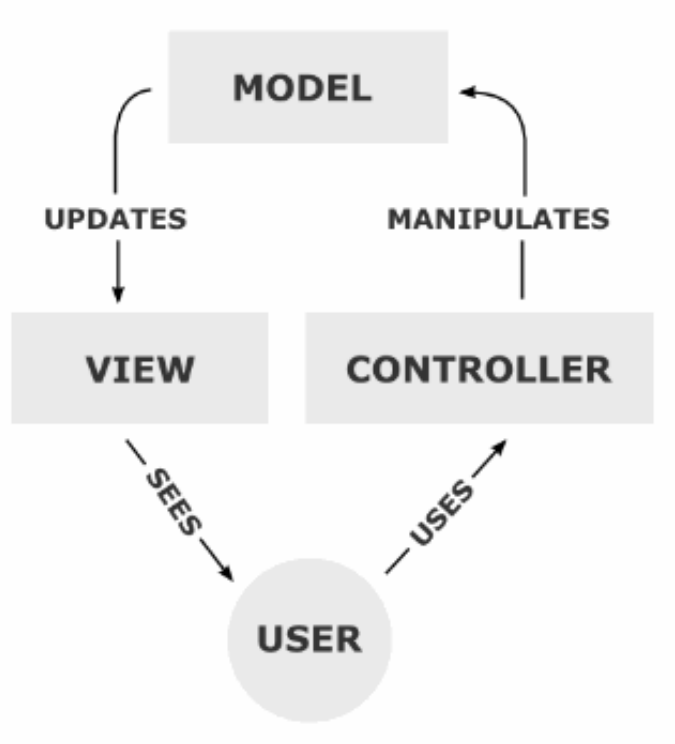
```
class MainWindow(QWidget):

    def __init__(self):
        super(MainWindow, self).__init__()
        self.setWindowTitle('Count Button')
        self.count = 0

        self.count_label = QLabel(str(self.count))
        self.inc_btn = QPushButton('Increment Count')
        self.inc_btn.clicked.connect(self.inc_count)

        self.vbox = QVBoxLayout()
        self.vbox.addWidget(self.count_label)
        self.vbox.addStretch(1)
        self.vbox.addWidget(self.inc_btn)

        self.setLayout(self.vbox)
```

See counter.py

Model-View-Controller

Complex GUI applications organized using MVC pattern:

- Model holds the data from the domain model
- View displays information to the user
- Controller coordinates between the model and the view



QListView and QStandardItemModel

In (Py)Qt, as in other GUI frameworks, view and controller are combined.

Here, a `QListView` displays the data contained in a `QStandardItemModel`.

```
self.list_view = QListView()
self.list_model = QStandardItemModel(self.list_view)
self.list_view.setModel(self.list_model)
```

See todo.py

Menus and Toolbars

```
exit = QAction("Exit", self)
exit.setShortcut("Ctrl+Q")
exit.setStatusTip("Exit quitter")
exit.triggered.connect(qApp.quit)

menu_bar = self.menuBar()
menu_bar.setNativeMenuBar(False)

file_menu = menu_bar.addMenu("&File")
file_menu.addAction(exit)
```

See quitter.py

Standard Dialog Boxes

```
file_name, filter = \
    QFileDialog.getOpenFileName(self, "Open file", ".",
                                "All files (*);;CSV Files (*.csv)")
```

See csv$_{gui}$.py

Examples

- helloqt.py
- label.py
- button.py
- message$_{\text{box}}$.py
- quitter.py
- counter.py