Creating GUIs with PyQT

John Strickler

Version 1.0, January 2025

Table of Contents

Chapter 1: PyQt

Objectives

- Explore PyQt programming
- Understand event-driven programming
- Code a minimal PyQt application
- Use the Qt designer to create GUIs
- Wire up the generated GUI to event handlers
- Validate input
- Use predefined dialogs
- Design and use custom dialogs

What is PyQt?

- Python Bindings for Qt library
- Qt written in C++ but ported to many languages
- Complete GUI library
- Cross-platform (OS X, Windows, Linux, et al.)
- Hides platform-specific details

PyQt is a package for Python that provides binding to the generic Qt graphics programming library. Qt provides a complete graphics programming framework, and looks "native" across various platforms. In addition to graphics components, it includes database access and many other tools.

It hides the platform-specific details, so PyQt programs are portable.

Matplotlib can be integrated with PyQT for data visualizations.

Event Driven Applications

- Application starts event loop
- When event occurs, goes to handler function, then back to loop
- Terminate event ends the loop (and the app)

GUI programs are different from conventional, procedural applications. Instead of the programmer controlling the order of execution via logic, the user controls the order of execution by manipulating the GUI.

To accomplish this, starting a GUI app launches an event loop, which "listens" for user-generated events, such as key presses, mouse clicks, or mouse motion. If there is a method associated with the event (AKA "event handler") (AKA "slot" in PyQt), the method is invoked.

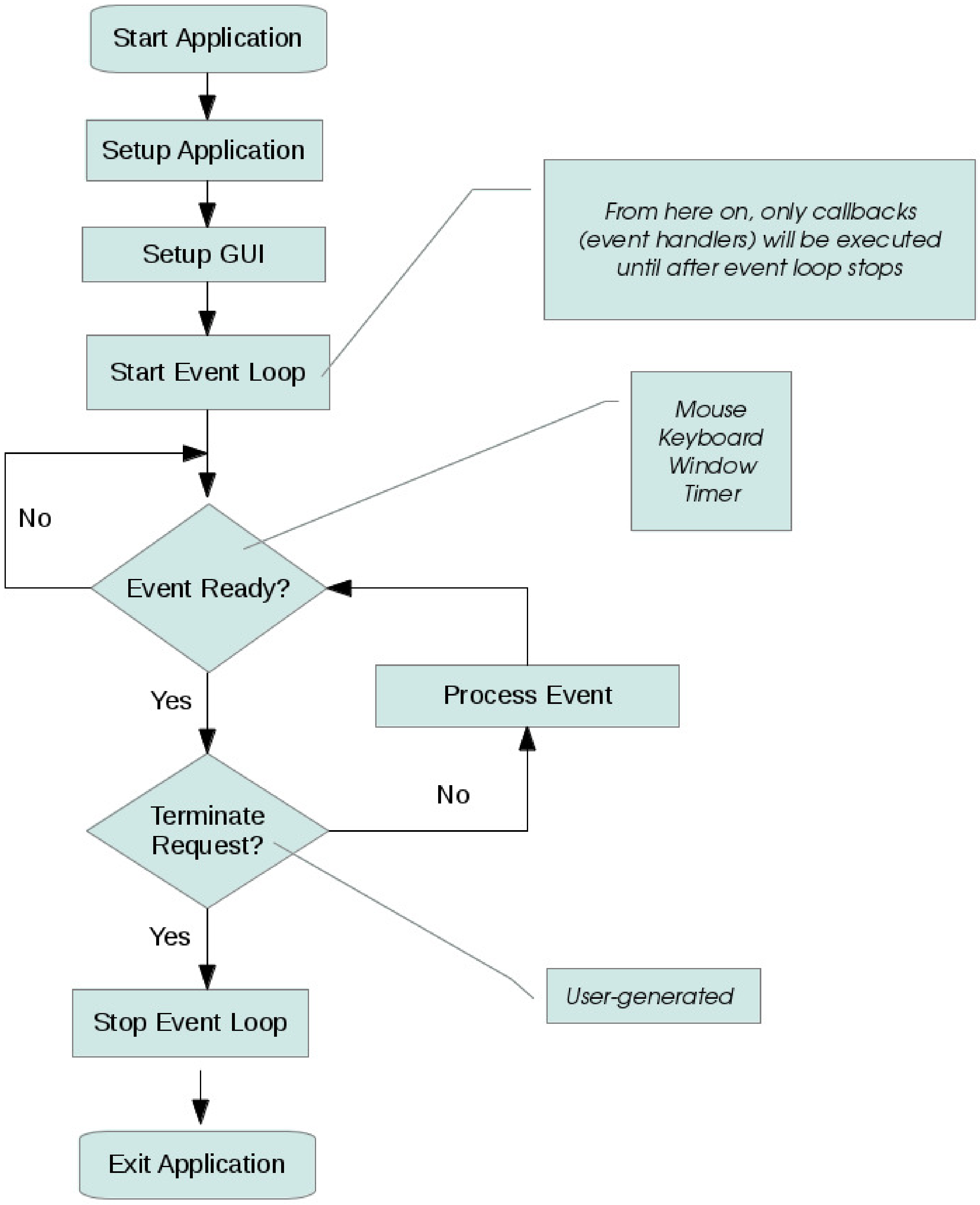
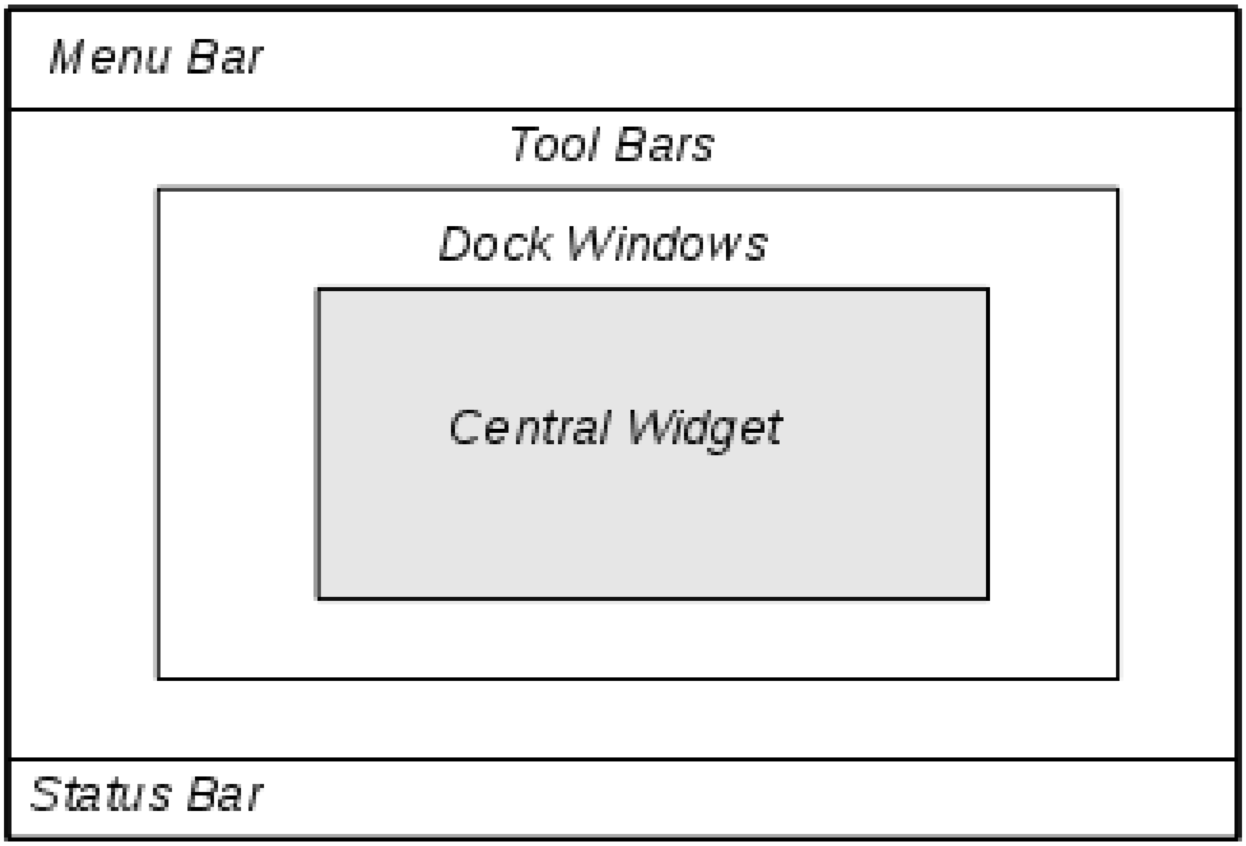


Figure 1. GUI Application Flow Chart

External Anatomy of a PyQt Application



The main window widget has several predefined areas:

- The menu bar area contains the usual File, Edit, and other standard menus.
- The tool bar areas can contain any tool bar buttons.
- The dock windows area contains any docked windows, which can be docked in any of the doc areas, or which can float freely. They have their own title bar with close, minimize and maximize buttons.
- The status bar can contain any other widgets, typically labels, but anything is fair game.
- None of the above are required, and if not present will not take up any screen space.
- The central widget is the main widget of the application. It is typically a QWidget layout object such as VBoxLayout, HBoxLayour, or GridLayout.

Internal Anatomy of a PyQt Application

- Extend (subclass) QmainWindow
- Call show() on main class

The normal (and convenient) approach is to subclass QMainWindow to create a custom main window for your application. Within this class, ***self*** is the main window of the application. You can attach widgets to and call setup methods from self.

QApplication is an object which acts as the application itself. You need to create a QApplication object and pass the command line arguments to it. To start your program call the exec_() method on your application object, after calling **show()** on the main window to make it visible.

Example

qt5/qt_hello.py

```
#!/usr/bin/env python
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication, QLabel # Standard PyQt5 imports

class HelloWindow(QMainWindow): # Main class inherits from QMainWindow to have normal
application behavior

    def __init__(self):
        super().__init__()
        self._label = QLabel("Hello PyQt5 World")
        self.setCentralWidget(self._label)

if __name__ == "__main__":
    app = QApplication(sys.argv)  # These 4 lines are always required. Only the name of
the main window object changes.
    main_window = HelloWindow()
    main_window.show()
    sys.exit(app.exec_())
```

Using designer

- GUI for building GUIs
- Builds applications, dialogs, and widgets
- Outputs generic XML that describes GUI
- pyuic5 (or pyuic4) generates Python module from XML
- Import generated module in main script

The **designer** tool makes it fast and easy to generate any kind of GUI.

To get started with designer, choose **menu:[File>New…]** from the File menu.

Select Main Window. (You can also use designer to create dialogs and widgets). This will create a blank application window. It will already have a menu bar and a status bar. It is ready for you to drag layouts and widgets as needed.

Be sure to change the objectName property of the QMainWindow object in the Property Editor. This (with "Ui_" prefixed) will be the name of the GUI class generated by pyuic4.

To set the title of your application, which will show up on the title bar, select the QmainWindow object in the Object Inspector, then open the QWidget group of properties in the Property Editor. Enter the title in the windowTitle property.

Be sure to give your widgets meaningful names, so when you have to use them in your main program, you know which widget is which. A good approach is a short prefix that describes the kind of widget (such as "bt" for QPushButton or "cb" for QComboBox) followed by a descriptive name. Good examples are 'bt_open_file', 'cb_select_state', and 'lab_hello'. There are no standard prefixes, so use whatever makes sense to you.

You can just drag widgets onto the main window and position them, but in general you will use layouts to contain widgets.

For each of the example programs, the designer file (.ui) and the generated module are provided in addition to the source of the main script.

PyQt designer

Designer-based application workflow

- Using designer
  - Create GUI
  - Name MainWindow Hello2 (for example)
  - Save from designer as hello2.ui
- Using pyuic5 (or pyuic4)
  - generate ui_hello2 .py from hello2.ui
- Using your IDE
  - Import PyQt5 widgets
  - Subclass QMainWindow
  - Add instance of Ui_Hello2 to main window
  - Call setupUi() from Ui_Hello2
  - Instantiate main window and call show()
- Start application

Example

qt5/qt_hello2.py

```
#!/usr/bin/env python

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_hello2 import Ui_Hello2  # import generated interface

class Hello2Main(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        # Set up the user interface generated from Designer.
        self.ui = Ui_Hello2()  # Attribute name does not have to be "ui"
        self.ui.setupUi(self)  # Create the widgets

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = Hello2Main()
    main.show()
    sys.exit(app.exec_())
```

Naming conventions

- Keep names consistent across applications
- Use "application name" throughout
- Pay attention to case

Using consistent names for the main window object can pay off in several ways. First, you'll be less confused. Second, you can write scripts or IDE macros to generate Python code from the designer output. Third, you can create standard templates which contain the boilerplate PyQt code to set up and run the GUI.

If application name is **Spam**:

- set **QMainWindow** object name to **Spam**
- set **windowTitle** to **Spam** (or as desired)
- Save design as `spam.ui`
- Redirect output of `pyuic5 spam.ui` to `ui_spam.py`
- In main program, use this import
  - `from ui_spam import Ui_Spam`

The file and object names are not required to be the same, or even similar. You can name any of these anything you like, but consistency can really help simplify code maintenance.

In `EXAMPLES/qt5` there is a file called `qt5_template.py` that you can copy, replacing `AppName` and `appname` with your app's name. This contains all of the required code for a normal Qt5 app.

Common Widgets

- `QLabel`, `QPushButton`, `QLineEdit`, `QComboBox`
- There are many more

The Qt library has many widgets; some are simple, and some are complex. Some of the most basic are `Qlabel`, `QPushButton`, `QLineEdit`, and `QComboBox`.

`QLabel` is a widget with some text. `QPushButton` is just a clickable button. `QLineEdit` is a one-line entry blank. `QComboBox` shows a list of values, and allows a new value to be entered.

`QLineEdit` widgets are typically paired with `QLabel` widgets. Using the property editor in designer, set the buddy property of the `QLabel` to be the matching `QLineEdit`. This allows the accelerator (specified with &letter in the label text) of the label to place focus on the paired widget.

Table 1. Common PyQt Widgets

Widget	Description
`QLabel`	Display non-editable text, image, or video
`QLineEdit`	Single line input field
`QPushButton`	Clickable button with text or image
`QRadioButton`	Selectable button; only one of a radio button group can be pushed at a time
`QCheckBox`	Individual selectable box
`QComboBox`	Dropdown list of items to select; allows new entry to be added
`QSpinBox`	Text box for integers with up/down arrow for incrementing/decrementing
`QSlider`	Line with a movable handle to control a bounded value
`QMenuBar`	A row of `QMenu` widgets displayed below the title bar
`QMenu`	A selectable menu (can have sub-menus)
`QToolBar`	Panel containing buttons with text, icons or widgets
`QInputDialog`	Dialog with text field plus OK and Cancel buttons
`QFontDialog`	Font selector dialog
`QColorDialog`	Color selector dialog
`QFileDialog`	Provides several methods for selecting files and folders for opening or saving
`QTab`	A tabbed window. Only one `QTab` is visible at a time
`QStacked`	Stacked windows, similar to `QTab`
`QDock`	Dockable, floating, window
`QStatusBar`	Horizontal bar at the bottom of the main window for displaying permanent or temporary information. May contain other widgets
`QList`	Item-based interface for updating a list. Can be multiselectable
`QScrollBar`	Add scrollbars to another widget
`QCalendar`	Date selector

Example

qt5/qt_commonwidgets.py

```
#!/usr/bin/env python

import sys
from PyQt5.QtWidgets import QMainWindow, QApplication

from ui_commonwidgets import Ui_CommonWidgets

class CommonWidgetsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_CommonWidgets()
        self.ui.setupUi(self)

        for k,v in (('apple',1),('banana',2),('mango',3)):  # populate the combo box
            self.ui.cbFruits.insertItem(v,k,v) # populate the combo box

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = CommonWidgetsMain()
    main.show()
    app.exec_()
```

Layouts

- `QVBoxLayout` (vertical, like pancakes)
- `QHBoxLayout` (horizontal, like books on a shelf)
- `QGridLayout` (rows and columns, like a chess board)
- `QFormLayout` (2 columns)

Most applications use more than one widget. To easily organize widgets into rows and columns, use layouts. There are four layout types: QVBoxLayout, QHBoxLayout, QGridLayout, and QFormLayout. Drag layouts to your Deigner canvas to create the arrangement you need; of course they may be nested.

QVBoxlayout and QHBoxlayout lay out widgets vertical or horizontally. Widgets will automatically be centered and evenly spaced.

QGridLayout lays out widgets in specified rows and columns. QFormLayout is a 2-column-wide grid, for labels and input widgets.

Layouts can be resized; widgets attached to them will grow and shrink as needed (by default – all PyQt behavior can be changed in the Property editor, or programmatically).

Example

qt5/qt_layouts.py

```
#!/usr/bin/env python

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_layouts import Ui_Layouts

class LayoutsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Layouts()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = LayoutsMain()
    main.show()
    sys.exit(app.exec_())
```

Selectable Buttons

- QRadioButton, QCheckBox
- Can be grouped

There are two kinds of selectable buttons. QRadioButtons are used in a group, where only one of the group can be checked. QCheckBoxes are individual, and can be checked or unchecked.

By default, radio buttons are auto-exclusive – all radio buttons that have the same parent are grouped. If you need more than one group of radio buttons to share a parent, use QButtonGroup to group them.

Use the isChecked() method to determine whether a button has been selected. Use the checked property to mark a button as checked.

Example

qt5/qt_selectables.py

```
#!/usr/bin/env python
import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_selectables import Ui_Selectables

class SelectablesMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Selectables()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    main = SelectablesMain()
    main.show()
    sys.exit(app.exec_())
```

Actions and Events

- Widgets have predefined actions that can be handled
- Handler is ordinary method
- Use widget.action.connect(method)
- Pass in function object; don’t call function
- Action is clicked, triggered, etc.
- Event names are predefined; override in main class

An event is something that changes in a GUI app, such as a mouse click, mouse movement, key press (or release), etc. To do something when an event occurs, you associate a function with an *action*, which represents the event. Actions are verbs that end with -ed, such as clicked, connected, triggered, etc.

To add an action to a widget, use the connect() method of the appropriate action, which is an attribute of the widget. For instance, if you have a QPushButton object pb, you can set the action with pb.clicked.connect(method).

Other events can be handled by using implementing predefined methods, such as keyPressEvent. Event handlers are passed the event object, which has more detail about the event, such as the key pressed, the mouse position, or the number of mouse clicks.

To get the widget that generated the event (AKA the sender), use self.sender() in the handler function.

If the action needs parameters, the simplest thing to do is use a lambda function as the handler (AKA slot), which can then call some other function with the desired parameters. NOTE: Actions and events are also called "signals" and "slots" in Qt.

Example

qt5/qt_events.py

```
#!/usr/bin/env python

import sys
import types

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_events import Ui_Events

def fprint(*args):
    """ print and flush the output buffer so text
        shows up immediately
    """
    print(*args)
    sys.stdout.flush()

class EventsMain(QMainWindow):
    FRUITS = dict(A='Apple', B='Banana', C='Cherry', D='Date')

    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Events()
        self.ui.setupUi(self)

        # set the File->Quit handler
        self.ui.actionQuit.triggered.connect(self.close) # Add an event handler callback
for when Quit is selected from the the FIle menu

        # set the Edit->Clear Name Field handler
        self.ui.actionClear_name_field.triggered.connect(self._clear_field)

        # use the same handler for all 4 buttons
        self.ui.pb_A.clicked.connect(self._mkfunc('red',self.ui.pb_A))  # Add a handler
for button A
        self.ui.pb_B.clicked.connect(self._mkfunc('blue',self.ui.pb_B))
        self.ui.pb_C.clicked.connect(self._mkfunc('yellow',self.ui.pb_C))
        self.ui.pb_D.clicked.connect(self._mkfunc('purple',self.ui.pb_D))

        self.setup_mouse_move_event_handler()

        self.ui.checkBox.toggled.connect(self._toggled)
        self.ui.checkBox.clicked.connect(self._clicked)
    def setup_mouse_move_event_handler(self):
        def mme(self, mouse_ev):
            self.ui.statusbar.showMessage("Motion: {0},{1}".format(   # Update status bar
                mouse_ev.x(), mouse_ev.y(), 0)  # 2nd param is timeout
            )
        # add method instance to label dynamically
        self.ui.label.mouseMoveEvent = types.MethodType(mme, self)

    def keyPressEvent(self, key_ev):
        """ Generated on keypresses """
        key_code = key_ev.key()   # Get the key that was pressed
        char = chr(key_code) if key_code < 128 else 'Special'  # See if it's a "normal"
key
        fprint("Key press: {0} ({1})".format(key_code, char))

    def mousePressEvent(self, mouse_ev):    # Overload mouse press event
        """ generated when mouse button is pressed """
        fprint("Press:", mouse_ev.x(), mouse_ev.y())



    def mouseReleaseEvent(self, mouse_ev):
        fprint("Release:", mouse_ev.x(), mouse_ev.y())

    def _toggled(self, mouse_ev):   # Handler when check box is toggled
        fprint("Toggle")

    def _clicked(self, mouse_ev):
        fprint("Click")

    def _checked(self, mouse_ev):
        fprint("Toggle")

    def _pushed(self):
        sender = self.sender()
        button_text = str(sender.text())
        if button_text in EventsMain.FRUITS:
            sender.setText(EventsMain.FRUITS[button_text])

    def _mkfunc(self, color, widget):  # Function factory to make button click event
handlers
        def pushed(stuff):
            button_text = str(widget.text())
            fprint("HI I AM BUTTON {0} and I AM {1}".format(button_text, color))
            if button_text in EventsMain.FRUITS:
                widget.setText(EventsMain.FRUITS[button_text])

        return pushed  # Function factory returns....a function
    def _clear_field(self):
        self.ui.leName.setText('')   # Update text on the LineEntry

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = EventsMain()
    main.show()
    sys.exit(app.exec_())
```

Signal/Slot Editor

- Event manager
  - Signal is event
  - Slot is handler
- Two ways to edit
  - Signal/Slot mode
  - Signal/Slot editor

The designer has an editor for connection signals (events) to builtin slots (handlers). You can select the widget that generates the event (emits the signal), and select which signal you want to handle. Then you can select the widget to receive the signal, and finally, select the method (on the receiving widget) to handle the event.

This is very handy for tying, for example, actionQuit.triggered to MainWindow.close()

It can’t be used for custom handlers. Do that in your main script.

Editing modes

- Widgets
- Signals/Slots
- Tab Order
- Buddies

By default **designer** is in *widget editing* mode, which allows dragging new widgets onto the window and arranging them. There are three other modes.

Signal/slot editing mode lets you drag and drop to connect signals (events generated by widgets) to slots (error handling methods). You can also use the separate signal/slot editor to do this.

Tab Order editing mode lets you set the tab order of the widgets in your interface. To do this, click on the widgets in the preferred order. You can right-click on widgets for other options. Tab order is the order in which the **Tab** key will traverse the widgets.

Buddies lets you pair labels with input widgets. Accelerator keys for the labels will jump to the paired input widgets.

Menu Bar

- Add menus and sub-menus to menu bar
- Add actions to sub-menus

In the Designer, you can just start typing on the menus in the menu bar to add menu items, separators, and sub-menus. To make the menus do something, see the examples in the previous topics. Note this section of code:

```
self.ui.actionQuit.triggered.connect(lambda:self.close())
       self.ui.actionClear_name_field.triggered.connect(self._clear_field)
```

This is how to attach a callback function to a menu item. By default, the designer will name menu choices based on the text in the menu. You can name the menu items anything, however.

You can also use the Signal/Slot editor in the Designer to handle menu events (signals) using builtin handlers (slots).

Status Bar

- Displays at bottom of main window

A status bar is a row at the bottom of the main window which can be used for text messages. A default status bar is automatically part of a GUI based on QmainWindow. It is named "statusbar".

To put a message on the status bar, use the showMessage() method of the status bar object. The first parameter is a string containing the message; the second parameter is the timeout in milliseconds. A timeout of 0 means display until overwritten. To clear the message, use either removeMessage(), or display an empty string.

Example

qt5/qt_statusbar.py

```
#!/usr/bin/env python
import sys
from PyQt5.QtWidgets import QMainWindow, QApplication
from statusbar_ui import Ui_StatusBar

class StatusBarMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self._count = 0

        # Setup the user interface from Designer.
        self.ui = Ui_StatusBar()
        self.ui.setupUi(self)

        self.ui.btPushMe.clicked.connect(self._pushed)
        self.ui.btPullYou.clicked.connect(self._pulled)
        self._update_statusbar(0) # do initial status bar update

    def _pushed(self, ev):
        self._update_statusbar(1)

    def _pulled(self, ev):
        self._update_statusbar(-1)

    def _update_statusbar(self, value):
        self._count += value
        msg = "Count is " + str(self._count)
        self.ui.statusbar.showMessage(msg, 0) # show message, 0 means no timeout, >= 0
means timeout in seconds


if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = StatusBarMain()
    main.show()
    app.exec()
```

Forms and validation

- Use form layout
- Validate input
- Use form data

PyQt makes it easy to create fill-in forms. Use a form layout (QFormLayout) to create a two-column form. Labels go in the left column and an entry widget goes in the right column. The entry widget can be any of a variety of widget types.

The Line Edit (QLineEdit) widget is typically used for a single-line text entry. You can add validators to this widget to accept or reject user input.

There are three kinds of validators — QRegExpValidator, QIntValidator, and QDoubleValidator. To use them, create an instance of the validator, and attach it to the Line Edit widget with the `setValidator()` method.

for QRexExpValidator, you need to create a QRexExp object to pass to it.

Example

qt5/qt_validators.py

```
#!/usr/bin/env python

import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5.QtGui import QRegExpValidator, QIntValidator
from PyQt5.QtGui import QDoubleValidator
from PyQt5.QtCore import QRegExp

from ui_validators import Ui_Validators

class ValidatorsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        # Set up the user interface from Designer.
        self.ui = Ui_Validators()

        self.ui.setupUi(self)
        self._set_validators()
        self.ui.bt_save.clicked.connect(self._save_pushed)

    def _set_validators(self):
        # Set up the validators (could be in separate function or module)
        reg_ex = QRegExp(r"[A-Za-z0-9]{1,10}") # create Qt regular expression object
(note use of raw string)
        val_alphanum = QRegExpValidator(reg_ex, self.ui.le_alphanum)  # create regex
validator from regex object
        self.ui.le_alphanum.setValidator(val_alphanum) # attach validator to line entry
field

        reg_ex = QRegExp(r"[a-z ]{0,30}") # create Qt regular expression object (note use
of raw string)
        val_lcspace = QRegExpValidator(reg_ex, self.ui.le_lcspace)  # create regex
validator from regex object
        self.ui.le_lcspace.setValidator(val_lcspace) # attach validator to line entry
field

        val_nums_1_100 = QIntValidator(1, 100, self.ui.le_nums_1_100)  # create integer
validator
        self.ui.le_nums_1_100.setValidator(val_nums_1_100)  # attach validator to line
entry field

        val_float = QDoubleValidator(0.0, 20.0, 2, self.ui.le_float)  # create double
(large float) validator
        self.ui.le_float.setValidator(val_float)  # attach validator to line entry field

    def _save_pushed(self):
        alphanum = self.ui.le_alphanum.text()
        lcspace = self.ui.le_lcspace.text()
        nums = self.ui.le_nums_1_100.text()
        fl = self.ui.le_float.text()
        msg = '{}/{}/{}/{}'.format(alphanum, lcspace, nums, fl)
        self.ui.statusbar.showMessage(msg, 0) # display valid date on status bar

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = ValidatorsMain()
    main.show()
    sys.exit(app.exec_())
```

Using Predefined Dialogs

- Predefined dialogs for common tasks
- Files, Messages, Colors, Fonts, Printing
- Use static methods for convenience.

PyQt defines several standard dialogs for common GUI tasks. The following chart lists them. Some of the standard dialogs can be invoked directly; however, most also provide some convenient static methods that provide more fine-grained control. These static methods return an appropriate value, typically a user selection.

Example

qt5/qt_standard_dialogs.py

```
#!/usr/bin/env python
import sys
import os



from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog, QColorDialog,
QErrorMessage, QInputDialog
from PyQt5.QtGui import QColor

from ui_standard_dialogs import Ui_StandardDialogs

class StandardDialogsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_StandardDialogs()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        # Connect up the buttons.
        self.ui.btFile.clicked.connect(self._choose_file)  # setup buttons to invoke
builtin dialogs
        self.ui.btColor.clicked.connect(self._choose_color)
        self.ui.btMessage.clicked.connect(self._show_error)
        self.ui.btInput.clicked.connect(self._get_input)
           # self.ui.BUTTON_NAME.clicked.connect(self._pushed)
    def _choose_file(self):
        full_path, _ = QFileDialog.getOpenFileName(self, 'Open file', os.getcwd()) #
invoke open-file dialog (starts in current directory); returns tuple of selected file
path and an empty string
        file_name = os.path.basename(full_path)
        self.ui.statusbar.showMessage("You chose: " + file_name) # update statusbar with
chosen filename

    def _choose_color(self):
        result = QColorDialog.getColor() # invoke color selector dialog and return result
        self.ui.statusbar.showMessage(
            "You chose #{0:02x}{1:02x}{2:02x} ({0},{1},{2})".format(
                result.red(),  # result has methods to retrieve color values
                result.green(),
                result.blue()
            )
        )



    def _show_error(self):
        em = QErrorMessage(self)  # invoke error message dialog with specified message
        em.showMessage("There is a problem")
        self.ui.statusbar.showMessage('Diplaying Error')

    def _get_input(self):
        text, ok = QInputDialog.getText(self, 'Input Dialog',
            'Enter your name:')  # invoke input dialog with prompt; returns entered text
and boolean flag -- True if user pressed OK, False if user pressed Cancel
        if ok:
            self.ui.statusbar.showMessage("Your name is " + text)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = StandardDialogsMain()
    main.show()
    sys.exit(app.exec_())
```

Table 2. Standard Dialogs (with Convenience Methods)

Dialog	Description
`QColorDialog.getColor`	Select a color
`QErrorMessage.showMessage`	Display an error messages

`QFileDialog.getExistingDirectory`
`QFileDialog.getOpenFileName`
`QFileDialog.getOpenFileNameAndFilter`
`QFileDialog.getOpenFileNames`
`QFileDialog.getSaveFileName`
`QFileDialog.getSaveFileNameAndFilter`
	Select files or folders

`QFontDialog`	Select a font

`QInputDialog.getText`
`QInputDialog.getInteger`
`QInputDialog.getDouble`
`QInputDialog.getItem`
	Get input from user

`QMessageBox`	Display a modal message
`QPageSetupDialog`	Select page-related options for printer
`QPrintPreviewDialog`	Display print preview
`QProgressDialog`	Display a progress windows
`QWizard`	Guide user through step-by-step process

Tabs

- Use a QTab Widget
- In designer under "Containers"/a
- Each tab can have a name

A QTab Widget contains one or more tabs, each of which can contain a widget, either a single widget or some kind of container.

As usual, tabs can be created in the designer. You should give each tab a unique name, so that in your main code you can access them programmatically.

Drag a Tab Widget to your application and place it. Then you can add more tabs by right-clicking on a tab and selecting Insert Page. You can then go to the properties of the tab widget and set the properties for the currently select tab. Select other tabs and change their properties as appropriate.

The actual tabs can be on any of the 4 sides of the tab widget, and they can be left-justified, right-justified, or centered. In addition, you can modify the shape of the tabs, and of course the color and font of the labels.

Whichever tab is selected when you generate the UI file will be selected when you start your application.

The QStacked widget is similar to QTab, but can stack *any* kind of widget.

Example

qt5/qt_tabs.py

```
#!/usr/bin/env python
import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from ui_tabs import Ui_Tabs

class TabsMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)

        self.ui = Ui_Tabs()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())
        self.ui.actionA.triggered.connect(lambda: self._show_tab('A'))
        self.ui.actionB.triggered.connect(lambda: self._show_tab('B'))
        self.ui.actionC.triggered.connect(lambda: self._show_tab('C'))

    def _show_tab(self, which_tab):
        if which_tab == 'A':
            self.ui.tabWidget.setCurrentIndex(0)  # choose tab programmatically
            self.ui.labA.setText('Aardvark')  # set text on label widget on tab
        elif which_tab == 'B':
            self.ui.tabWidget.setCurrentIndex(1)  # choose tab programmatically
            self.ui.labB.setText('Bonobo')  # set text on label widget on tab
        elif which_tab == 'C':
            self.ui.tabWidget.setCurrentIndex(2)  # choose tab programmatically
            self.ui.labC.setText('Coatimundi')  # set text on label widget on tab

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = TabsMain()
    main.show()
    sys.exit(app.exec_())
```

Niceties

- Styling Text
- Tooltips

Fonts can be configured via the designer. Choose any widget in the **Object Inspector**, then search for the **font** property in the **Property Editor**. You can change the font family, the point size, and weight and slant of the text.

You can further style a widget by specifying a string containing valid CSS (cascading style sheet). In the CSS, the selectors are `ObjectType` for all objects of that type, or `ObjectType#object_name`. To style a particular object.

If you have many styles, it makes sense to read in a stylesheet from a file.

```
widget.setStyleSheet('QPushButton {color: blue}')
widget.setStyleSheet('QPushButton#btn_start {color: red}')
widget.setStyleSheet(open("myapp.css").read())
```

Tooltips are added by calling `.setToolTip("tool tip text")` on any widget.

Tooltips can easily be added via **designer**. Search **Property Editor** for the **toolTip** property and type in the desired text.

Working with Images

- Display images via QLabels
- Create a QPixMap of the image
- Assign pixmap to label

To display an image, create a Pixmap object from the graphics file. Then assign the pixmap to a QLabel. The image may be scaled or resized.

The images in the example program are scaled to fit the original label. This is why they are distorted.

Example

qt5/qt_images.py

```
#!/usr/bin/env python
import sys

from PyQt5.QtWidgets import QMainWindow, QApplication
from PyQt5.QtGui import QPixmap
from ui_images import Ui_Images

class ImagesMain(QMainWindow):

    def __init__(self):
        QMainWindow.__init__(self)

        # Set up the user interface from Designer.
        self.ui = Ui_Images()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        self.ui.actionPictureA.triggered.connect(
            lambda: self._show_picture('A')
        )
        self.ui.actionPictureB.triggered.connect(
            lambda: self._show_picture('B')
        )
        self.ui.actionPictureC.triggered.connect(
            lambda: self._show_picture('C')
        )

    def _show_picture(self, which_picture):
        if which_picture == 'A':
            image_file = 'apple.png'  # select image
            label = self.ui.labA  # select label for image
        elif which_picture == 'B':
            image_file = 'banana.jpg'
            label = self.ui.labB
        elif which_picture == 'C':
            image_file = 'cherry.jpg'
            label = self.ui.labC

        img = QPixmap('../../DATA/' + image_file)  # create QPixMap from image
        label.setPixmap(img)  # assign pixmap to label
        label.setScaledContents(True)  # scale picture
```

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = ImagesMain()
    main.show()
    sys.exit(app.exec_())
```

Complete Example

This is an application that lets the user load a file of words, then shows all words that match a specified regular expression

Example

qt5/qt_wordfinder.py

```
#!/usr/bin/env python
import sys
import re

from PyQt5.QtWidgets import QMainWindow, QApplication, QFileDialog
from ui_wordfinder import Ui_WordFinder

class WordFinderMain(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)


        # Set up the user interface from Designer.
        self.ui = Ui_WordFinder()
        self.ui.setupUi(self)

        self.ui.actionQuit.triggered.connect(lambda:self.close())

        self.ui.actionLoad.triggered.connect(self._load_file)

        self.ui.lePattern.returnPressed.connect(self._search)

        # the following might be too time-consuming for large files
        # self.ui.lePattern.textChanged.connect(self._search)

        self.ui.btSearch.clicked.connect(self._search)

    def _load_file(self):
        file_name, _ = QFileDialog.getOpenFileName(
            self, 'Open file for matching', '.'
        )
        if file_name:
            with open(file_name) as F:
                self._words = [ line.rstrip() for line in F ]
            self._numwords = len(self._words)
            self.ui.teText.clear()

            self.ui.teText.insertPlainText(
                '\n'.join(self._words))

    def _search(self):
        pattern = str(self.ui.lePattern.text())
        if pattern == '':
            pattern = '.'
        rx = re.compile(pattern)

        self.ui.teText.clear()
        self.ui.lePattern.setEnabled(False)
        # self.lePattern.setVisible(False)
        count = 0
        for word in self._words:
            if rx.search(word):
                self.ui.teText.insertPlainText(word + '\n')
                count += 1
        self.ui.lePattern.setEnabled(True)
        #self.ui.lePattern.setVisible(True)
        self.ui.statusbar.showMessage(
            "Matched {0} out of {1} words".format(count,self._numwords),
            0
        )

if __name__ == '__main__':
    app = QApplication(sys.argv)
    main = WordFinderMain()
    main.show()
    sys.exit(app.exec_())
```

Chapter 1 Exercises

Exercise 1-1 (gpresinfo.py, ui_gpresinfo.py, gpresinfo.ui)

Using the Qt designer, write a GUI application to display data from the **presidentsqlite** or **presidentmysql** module. It should have at least the following components:

- A text field (use Entry) for entering the term number
- A Search button for retrieving data
- An Exit button

A widget or widgets to display the president’s information – you could use a Text widget, multiple Labels, or whatever suits your fancy.

Be creative.

For the ambitious Provide a combo box of all available presidents rather than making the user type in the name manually.

For the even more ambitious Implement a search function – let a user type text in a blank, and return a list of presidents which matches the text in either the first name or last name field. E.g., "jeff" would retrieve "Jefferson, Thomas", and "John" would retrieve "Adams, John", as well as "Kennedy, John", "Johnson, Lyndon" and so forth.

Index