Chapter 14

Graphical User Interfaces

So far, we have developed programs that interact with the user through the command line, where the user has to call a Python program by typing its name and adding the sequence of arguments.

Having a visual environment for the program variables and results extremely simplify the interaction between the user and the application. This kind of environments are known as a *Graphical User Interfaces (GUI)*. Graphical interfaces are present in various types of devices and platforms, such as web form or a smartphone application. Most, if not all, graphical user interface based applications use an *event management based architecture*. Applications operated by command line perform data input and output at specific times and circumstances established within the program. However, every time we interact with an application graphically, the program does not know beforehand when actions will occur. The user may enter data, press a key, move the mouse, or click on any widget within the interface. Therefore, the code must be written to respond to all these events. It is always possible to design a graphical interface from the beginning to the end by interacting directly with each pixel. Never the less, there are now optimized modules that provide generic graphical elements such as buttons, bars, text boxes, and calendars. These modules greatly facilitate the development of graphical interfaces. In this course, we will focus on the use of *PyQt*.

14.1 PyQt

PyQt is a Python library that includes several modules that help with the development of graphical interfaces, here we describe some of the modules included in *PyQt*:

- QtCore: includes non-GUI functionalities like file and directory management, time, and URLs, among others.
- QtGui: includes visual components such as buttons, windows, drop-down lists, etc.

- QtNetwork: provides classes to program graphical applications based on TCP/IP, UDP, and some other network protocols.
- QtXml: includes XML file handling functionalities.
- QtSvg: provides classes to display vector graphics files (SVG).
- QtOpenGL: contains functions for 3D and 2D graphics rendering using OpenGL.
- QtSql: includes functionalities for working with SQL databases.

To create a window we use the *QtGui* module. The first step is to build the application that will contain the window and all its elements or *Widgets*. This procedure is done through the `QApplication` class, which includes event loop, application initialization and closing, input arguments handling, among other responsibilities. For every application that *PyQt* uses, there is only one `QApplication` object regardless of the number of windows that this application has.

```python
# codes_1.py

from PyQt4 import QtGui


class MiForm(QtGui.QWidget):
    # The next line defines the window geometry.
    # Parameters: (x_top_left, y_top_left, width, height)
    self.setGeometry(200, 100, 300, 300)
    self.setWindowTitle('My First Window')  # Optional


if __name__ == '__main__':
    app = QtGui.QApplication([])
    form = MiForm()
    form.show()
    app.exec_()
```

The `QtGui.QWidget` class from which the `MyForm` class descends, is the base class for all objects in *PyQt*. The constructor for this class has no default parents, in which case corresponds to an empty window. We also have to define the window properties, so far only defined in memory. To display the window on the screen, we must use the show() method. Finally, the exec_() method executes the main loop to perform the event detection. The result of the above code corresponds to the clear interface as shown in Figure 14.1.

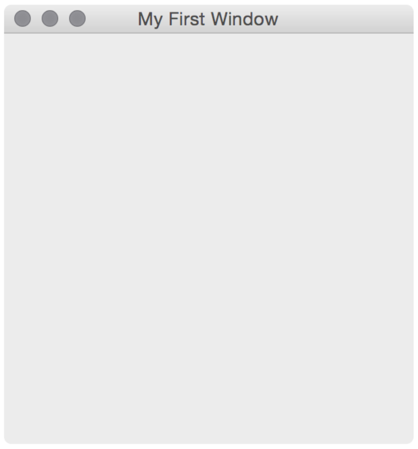


Figure 14.1: Example of an empty window generated by PyQt.

In *PyQt* there are useful objects or Widgets to control information input and output in graphical interfaces. These are labels and text boxes. Labels are used to display form variables or static strings. In *PyQt* these are represented by the QLabel class. Text boxes also allow text handling in the interface, primarily as a means of entering data into the form. In *PyQt* interface they are created by QLineEdit class:

```python
# codes_2.py

from PyQt4 import QtGui


class MiForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_GUI()

    def init_GUI(self):
        # Once the form is called, this method initializes the
        # interface and all of its elements and Widgets

        self.label1 = QtGui.QLabel('Text:', self)
        self.label1.move(10, 15)

        self.label2 = QtGui.QLabel('This label is modifiable', self)
        self.label2.move(10, 50)

        self.edit1 = QtGui.QLineEdit('', self)
        self.edit1.setGeometry(45, 15, 100, 20)

        # Adds all elements to the form
        self.setGeometry(200, 100, 200, 300)
        self.setWindowTitle('Window with buttons')


if __name__ == '__main__':
    app = QtGui.QApplication([])

    # A window that inherits from QMainWindow is created
    form = MiForm()
    form.show()
    app.exec_()
```

The code above clarifies how to use `Labels` and `LineEdits` within the GUI. Figure 14.2 shows the results.

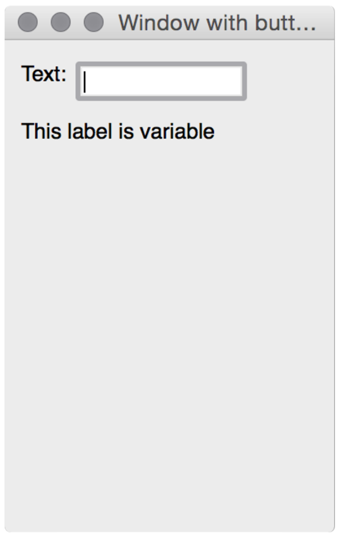


Figure 14.2: Example of a QLabel within a Widget.

PyQt also includes several useful graphical objects to control the interface. The most basic is the `PushButton(label, father, function)` element, which embeds a button in the window. The result generated by the code below is a window with a button as shown in the Figure 14.3.

```python
# codes_3.py

from PyQt4 import QtGui


class MyForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_GUI()

    def init_GUI(self):
        # Once the form is called, this method initializes the
        # interface and all of its elements and Widgets
        self.label1 = QtGui.QLabel('Text:', self)
        self.label1.move(10, 15)

        self.label2 = QtGui.QLabel('Write the answer here', self)
        self.label2.move(10, 50)

        self.edit1 = QtGui.QLineEdit('', self)
        self.edit1.setGeometry(45, 15, 100, 20)

        # The use of the & symbol at the start of the text within
        # any button or menu makes it so the first letter is shown
        # in bold font. This visualization may depend on the used
        # platform.
        self.button1 = QtGui.QPushButton('&Process', self)
        self.button1.resize(self.button1.sizeHint())
        self.button1.move(5, 70)

        # Adds all elements to the form
        self.setGeometry(200, 100, 200, 300)
        self.setWindowTitle('Window with buttons')
        self.show()
```

```python


if __name__ == '__main__':
    app = QtGui.QApplication([])

    # A window that inherits from QMainWindow is created
    form = MyForm()
    form.show()
    app.exec_()
```

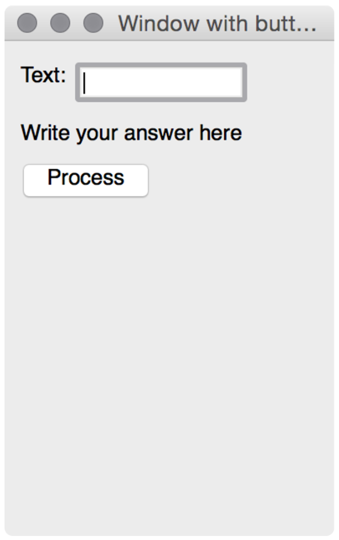


Figure 14.3: A simple window with a button.

Main Window

Windows created by `QWidget` correspond to windows that may contain other Widgets. *PyQt* offers a more complete type of window called `MainWindow`. It creates the standard skeleton of an application including a status bar, toolbar, and menu bar, as shown in the Figure 14.4.

The **status bar** allows us to display application status information. To create this, the `statusBar()` method (belongs to `QApplication` class) is used. Messages in the status bar are updated when the user interacts with the rest of the application. The **menu bar** is a typical part of a GUI-based application. It corresponds to a group of structured and logically grouped commands inside of menus. The **toolbar** provides quick access to the most frequently used commands. Finally, the **Central Widget** or core content area corresponds to the body of the window. It may contain any Widget in *QtGui*, as well as one of the forms created in the previous examples. To add this Widget or form to

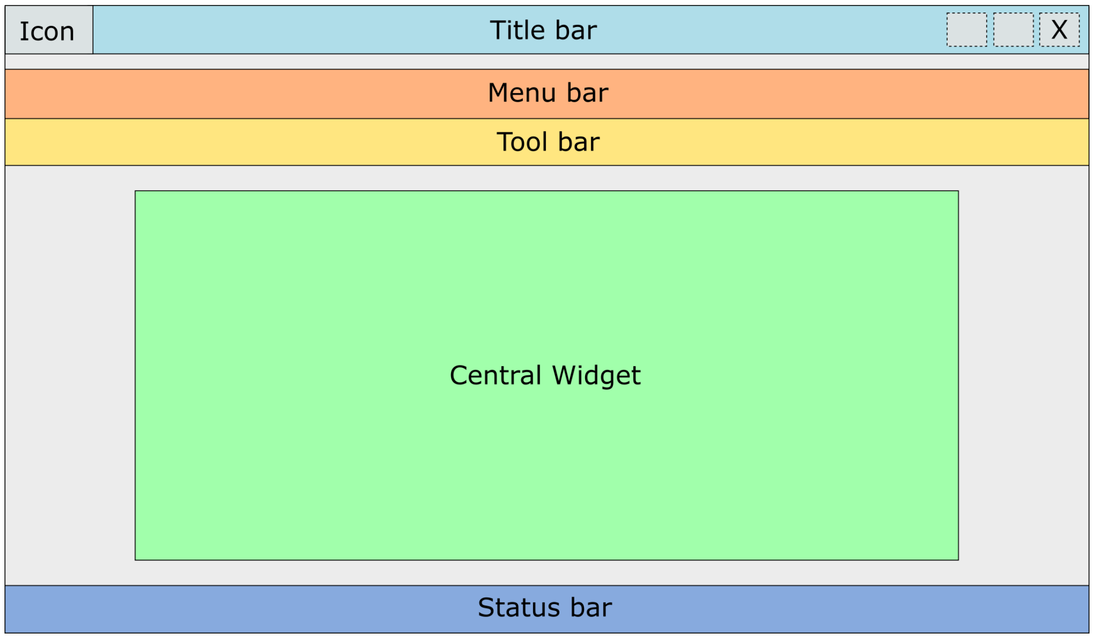


Figure 14.4: Diagram of the classic `MainWindow` layout.

the central Widget, the `setCentralWidget(Widget)` method must be used. The next example shows how to integrate the elements described in the main window:

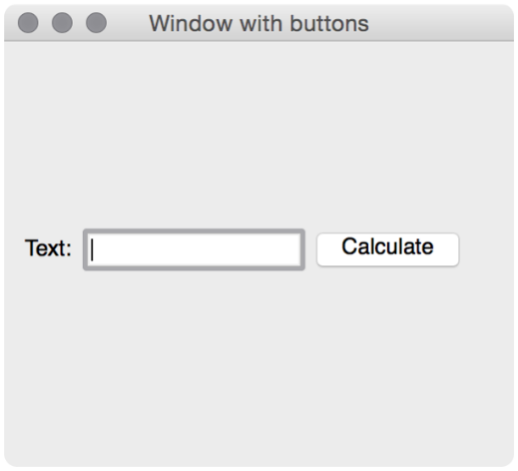
```python
# codes_4.py

from PyQt4 import QtGui


class MainForm(QtGui.QMainWindow):
    def __init__(self):
        super().__init__()

        # Configures window geometry
        self.setWindowTitle('Window with buttons')
        self.setGeometry(200, 100, 300, 250)

        # Action definitions
        see_status = QtGui.QAction(QtGui.QIcon(None), '&Change '
                                                       'Status', self)
        see_status.setStatusTip('This is a test item')
        see_status.triggered.connect(self.change_status_bar)

        exit = QtGui.QAction(QtGui.QIcon(None), '&Exit', self)
        # We can define a key combination to execute each command
        exit.setShortcut('Ctrl+Q')
        # This method shows the command description on the status bar
        exit.setStatusTip('End the app')
        # Connects the signal to the slot that will handle this event
        exit.triggered.connect(QtGui.qApp.quit)

        # Menu and menu bar creation
        menubar = self.menuBar()
        file_menu = menubar.addMenu('&File')  # first menu
        file_menu.addAction(see_status)
        file_menu.addAction(exit)

        other_menu = menubar.addMenu('&Other Menu')  # second menu

        # Includes the status bar
        self.statusBar().showMessage('Ready')

        # Sets the previously created form as the central widged
        self.form = MyForm()
        self.setCentralWidget(self.form)

    def change_status_bar(self):
        self.statusBar().showMessage('Status has been changed')


class MyForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_GUI()

    def init_GUI(self):
        self.label1 = QtGui.QLabel('Text:', self)
        self.label1.move(10, 15)

        self.label2 = QtGui.QLabel('Write the answer here', self)
        self.label2.move(10, 50)

        self.edit1 = QtGui.QLineEdit('', self)
        self.edit1.setGeometry(45, 15, 100, 20)

        self.button1 = QtGui.QPushButton('&Process', self)
        self.button1.resize(self.button1.sizeHint())
        self.button1.move(5, 70)

        self.setGeometry(200, 100, 200, 300)
        self.setWindowTitle('Window with buttons')
        self.show()

    def button_pressed(self):
        sender = self.sender()
        self.label3.setText('Signal origin: {0}'.format(sender.text()))
        self.label3.resize(self.label3.sizeHint())

    def button1_callback(self):
        self.label2.setText('{}'.format(self.edit1.text()))
        self.label2.resize(self.label2.sizeHint())


if __name__ == '__main__':
    app = QtGui.QApplication([])

    form = MainForm()
    form.show()
    app.exec_()
```

14.2 Layouts

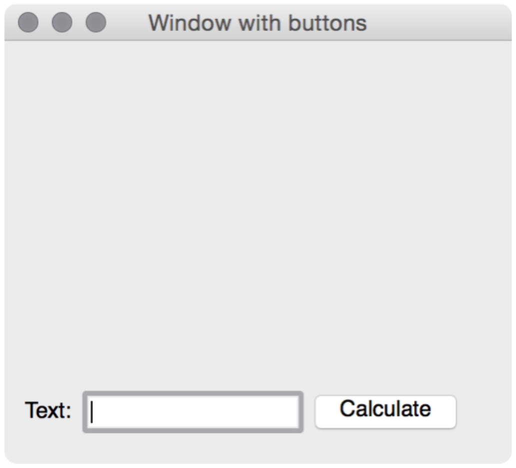
Layouts allow a more flexible and responsive way to manage and implement Widget distribution in the interface window. Each Widget's `move(x, y)` method allows absolute positioning of all objects in the window. However, it has limitations, such as; Widget position does not change if we resize the window, and Application's look will vary on different platforms or display settings.

To avoid redoing the window for better distribution, we use **box layouts**. Two basic types allow Widget horizontal and vertical alignment: `QtGui.QHBoxLayout()` and `QtGui.QVBoxLayout()`. In both cases, Widgets are distributed within the layout occupying all available space, even if we resize the window. Objects must be added to each layout by the `addWidget` method. Finally, the box layout must be loaded to the window as `self.setLayout()`. We can add the vertical alignment of objects by including the horizontal layout within a vertical layout. The following code shows an example of how to create a layout such that three Widgets are aligned in the bottom right corner. Figure 14.5 shows the output:

```python
# codes_5.py

from PyQt4 import QtGui


class MainForm(QtGui.QMainWindow):
    def __init__(self):
        super().__init__()

        # Window geometry
        self.setWindowTitle('Window with buttons')
        self.setGeometry(200, 100, 300, 250)

        self.form = MiForm()
        self.setCentralWidget(self.form)


class MiForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_GUI()

    def init_GUI(self):
        self.label1 = QtGui.QLabel('Text:', self)
        self.label1.move(10, 15)

        self.edit1 = QtGui.QLineEdit('', self)
        self.edit1.setGeometry(45, 15, 100, 20)

        self.button1 = QtGui.QPushButton('&Calculate', self)
        self.button1.resize(self.button1.sizeHint())

        # QHBoxLayout() and QVBoxLayout() are created and added to the
        # Widget list by using the addWidget() method. The stretch()
        # method includes a spacing that expands the layout towards
        #  the right and downwards.
        hbox = QtGui.QHBoxLayout()
        hbox.addStretch(1)
        hbox.addWidget(self.label1)
        hbox.addWidget(self.edit1)
        hbox.addWidget(self.button1)

        vbox = QtGui.QVBoxLayout()
        vbox.addStretch(1)
        vbox.addLayout(hbox)

        # The vertical layout contains the horizontal layout
        self.setLayout(vbox)


if __name__ == '__main__':
    app = QtGui.QApplication([])

    form = MainForm()
    form.show()
    app.exec_()
```



(a)



(b)

Figure 14.5: This figure shows the possible results after executing the code above. (a) Shows only using `QHBoxLayout`. (b) Shows using `QHBoxLayout` and `QVBoxLayout`.

PyQt includes a class to distribute widgets in a matrix-like grid within the window, called `QGridLayout()`. This type of layout divides the window space into rows and columns. Each Widget must be added to a cell in the grid by using the `addWidget(Widget, i, j)` method. For example, the following code shows an example to create a matrix similar to a mobile phone keypad buttons. The output can be seen in Figure 14.6.

```python
# codes_6.py

class MyForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_GUI()

    def init_GUI(self):
        # Creating the grid that will position Widgets in a matrix
        # like manner
        grid = QtGui.QGridLayout()
        self.setLayout(grid)

        values = ['1', '2', '3',
                  '4', '5', '6',
                  '7', '8', '9',
                    '*', '0', '#']

        positions = [(i, j) for i in range(4) for j in range(3)]

        for _positions, value in zip(positions, values):
            if value == '':
                continue

            # The * symbol allows unpacking _positions as
            # independent arguments
            button = QtGui.QPushButton(value)
            grid.addWidget(button, *_positions)

        self.move(300, 150)
        self.setWindowTitle('Calculator')
        self.show()
```

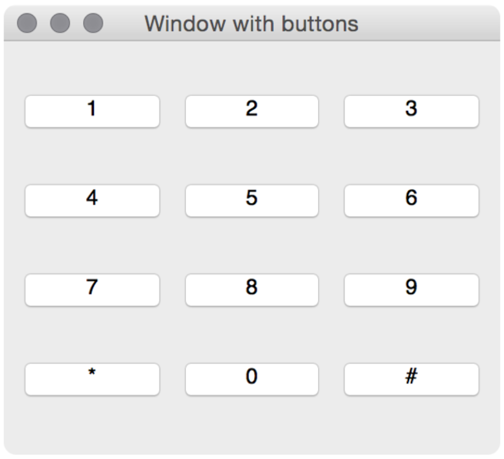


Figure 14.6: Example of numeric keypad using `QGridLayout` to organize the buttons in a grid.

14.3 Events and Signals

Graphical interfaces are applications focused mainly on handling events. This strategy allows detecting user actions on the interface asynchronously. The same application can generate events A. In *PyQt* these events are detected once the application enters the main loop started by the `exec_()` method. Figure 14.6 shows a flowchart comparison between a program with a linear structure and a program using GUI-based event handling. In this model there are three fundamental elements:

- The source of the event: corresponds to the object that generates the change of state or that generates the event
- The event object: the object that encapsulates the change of status through the event.
- The target object: the object to be notified of the status change

Under this model, the event source delegates the task of managing the event to the target object. *PyQt*, on its 4th version, uses the *signal* and *slot* mechanism to handle events. Both elements are used for communication between objects. When an event occurs, the activated object generates signals, and any of those signals calls a slot. Slots can be any callable Python object.

Below, we see a modification to the previous program so that it generates a call to the `boton1_callback()` function, after `button1` is pressed. This is accomplished using the event to send the signal. In the case of buttons, the signal corresponds to the *clicked* method. By using the `connect()` method, communication between objects involved in the event is set. This method receives a Python callable function, i.e., `boton1_callback` without `()`.

```python
# codes_7.py

class MyForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.init_GUI()

    def init_GUI(self):
        self.label1 = QtGui.QLabel('Text:', self)
        self.label1.move(10, 15)

        self.label2 = QtGui.QLabel('Write your answer here', self)
        self.label2.move(10, 50)

        self.edit1 = QtGui.QLineEdit('', self)
        self.edit1.setGeometry(45, 15, 100, 20)

        # Connecting button1 signal to other object
        self.button1 = QtGui.QPushButton('&Process', self)
        self.button1.resize(self.button1.sizeHint())
        self.button1.move(5, 70)
        # This object MUST be callable. self.button1_callback()
        # would not work.
        self.button1.clicked.connect(self.button1_callback)

        self.button2 = QtGui.QPushButton('&Exit', self)
        self.button2.clicked.connect(
            QtCore.QCoreApplication.instance().quit)
        self.button2.resize(self.button2.sizeHint())
        self.button2.move(90, 70)

        self.setGeometry(200, 100, 200, 300)
        self.setWindowTitle('Window with buttons')

    def button1_callback(self):
        # This method handles the event
        self.label2.setText(self.edit1.text())
```

14.4 Sender

Sometimes we need to know which of the objects on the form sent a signal. *PyQt* offers the `sender()` method. We can see an example by adding a new label to the form that will display the name of the widget that sends the signal:

```
# codes_8.py

def init_GUI(self):
    self.label1 = QtGui.QLabel('Text:', self)
    self.label1.move(10, 15)

    self.label2 = QtGui.QLabel('Write your answer here:', self)
    self.label2.move(10, 50)

    self.label3 = QtGui.QLabel('Signal origin:', self)
    self.label3.move(10, 250)
```

```

        self.edit1 = QtGui.QLineEdit('', self)
        self.edit1.setGeometry(45, 15, 100, 20)

        self.button1 = QtGui.QPushButton('&Process', self)
        self.button1.resize(self.button1.sizeHint())
        self.button1.move(5, 70)
        self.button1.clicked.connect(self.button1_callback)
        self.button1.clicked.connect(self.button_pressed)

        self.button2 = QtGui.QPushButton('&Exit', self)
        self.button2.clicked.connect(QtCore.QCoreApplication.instance().quit)
        self.button2.resize(self.button2.sizeHint())
        self.button2.move(90, 70)

        self.setGeometry(200, 100, 300, 300)
        self.setWindowTitle('Window with buttons.')
        self.show()


    def button_pressed(self):
        # This method registers the object sending the signal and shows it in
        # label3 by using the sender() method
        sender = self.sender()
        self.label3.setText('Signal origin: {0}'.format(sender.text()))
        self.label3.resize(self.label3.sizeHint())


    def button1_callback(self):
        self.label2.setText(self.edit1.text())
```

14.5 Creating Custom Signals

In *PyQt* it is also possible to define user-customized signals. In this case, we must create the object that will host the new signal. These signals are a subclass of `QtCore.QObject`. Within the object the new signal is created as an instance of the object `QtCore.pyqtSignal()`. Then, the signal and its handling functions if required, must be created in the form. The example below shows a simple way to generate a new signal that activates when one of the buttons is pressed. To emit the signal the `emit()` method inherited from `pyqtSignal()` is used:

```python
# codes_9.py

import sys
from PyQt4 import QtGui, QtCore


class MySignal(QtCore.QObject):
    # This class defines the new signal 'signal_writer'
    signal_writer = QtCore.pyqtSignal()


class MyForm(QtGui.QWidget):
    def __init__(self):
        super().__init__()
        self.initialize_GUI()

    def initialize_GUI(self):
        self.s = MySignal()
        self.s.signal_writer.connect(self.write_label)

        self.label1 = QtGui.QLabel('Label', self)
        self.label1.move(20, 10)
        self.resize(self.label1.sizeHint())

        self.setGeometry(300, 300, 290, 150)
        self.setWindowTitle('Signal Emitter')
        self.show()

    def mousePressEvent(self, event):
        # This method handles when any of the mouse buttons is pressed. It is
        # defined by default within the app. It can be overwritten according
        # to how the event should be handled in each app.
```

```python
            self.s.signal_writer.emit()

    def write_label(self):
        self.label1.setText('Mouse was clicked')
        self.label1.resize(self.label1.sizeHint())


def main():
    app = QtGui.QApplication(sys.argv)
    ex = MyForm()
    sys.exit(app.exec_())


if __name__ == '__main__':
    main()
```

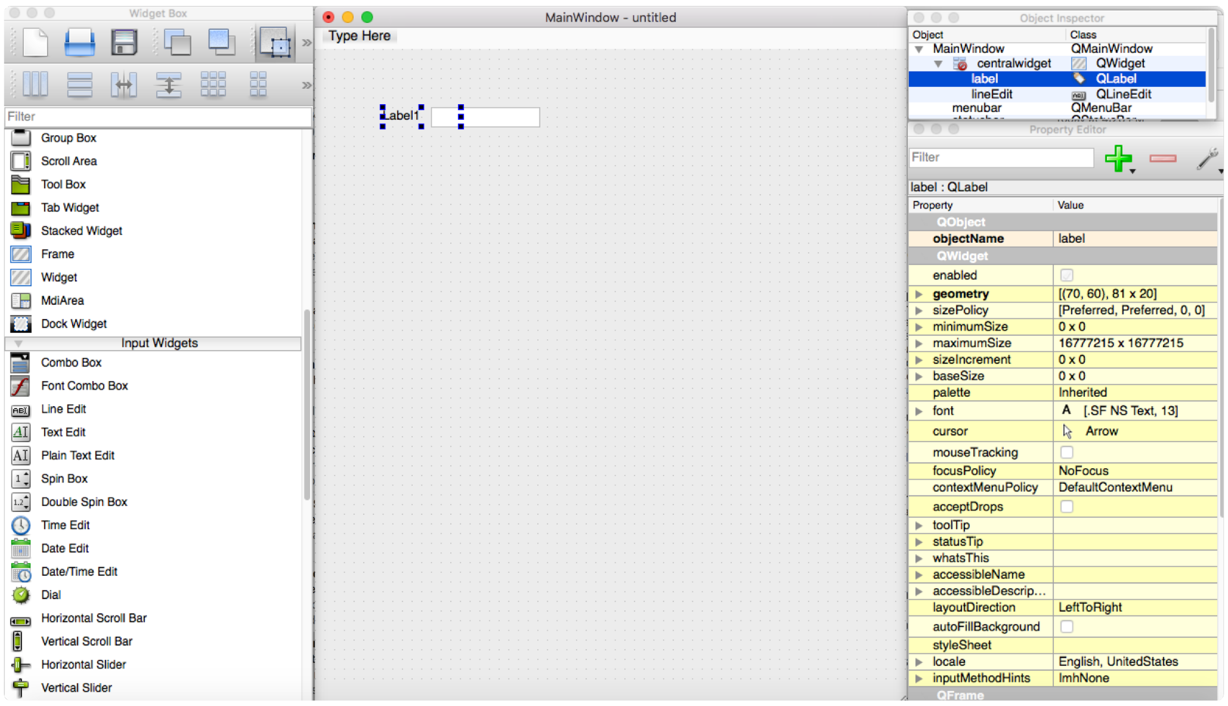
14.6 Mouse and Keyboard Events

Another way to generate events is through the keyboard and mouse. These events can be handled by overriding two methods defined in the `MainForm`: `mousePressEvent()` and `keyPressEvent()`.

```python
# codes_10.py

def keyPressEvent(self, event):
    self.statusBar().showMessage('Key pressed {}'.format(event.text()))


def mousePressEvent(self, *args, **kwargs):
    self.statusBar().showMessage('Mouse click')
```

14.7 QT Designer

When GUIs have few Widgets, it is easy to create them manually by adding each one with code. However, when the interface includes a larger number of objects, interactions or controls, the code gets longer and hard to maintain. Fortunately, *PyQt* provides a tool called *QT Designer* that allows building the graphical interface visually. *Qt Designer* allows to create each widget in the interface and also gives you control over all the properties of Widgets. Figure 14.7 shows the main view of *Qt Designer*.



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
label	QLabel
lineEdit	QLineEdit
menubar	QMenuBar

label : QLabel

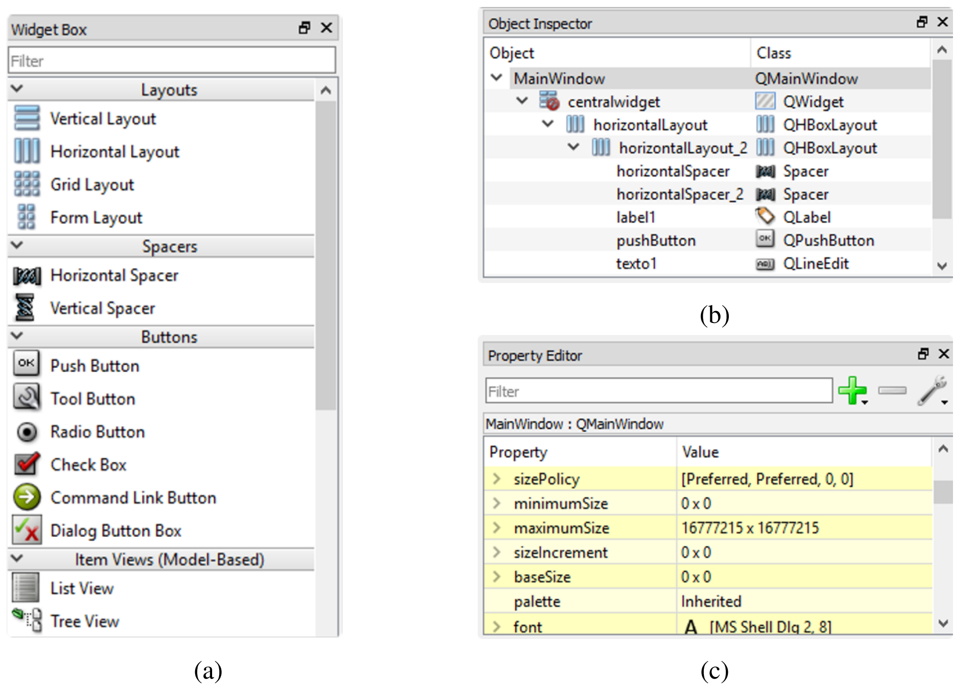
Property	Value
QObject	
objectName	label
QWidget	
enabled	✓
geometry	[(70, 60), 81 x 20]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [.SF NS Text, 13]
cursor	Arrow
mouseTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
toolTip	
statusTip	
whatsThis	
accessibleName	
accessibleDescrip…	
layoutDirection	LeftToRight
autoFillBackground	☐
styleSheet	
locale	English, UnitedStates
inputMethodHints	ImhNone
QFrame	

Figure 14.7: Example of *QtDesigner* application's main view.

All *QTDesigner*'s actions can be controlled from the application menu. There are four major sections in the working window. On the left side is the Widget Box containing all Widgets that can be added to the form, sorted by type. On the right side, we can find the Object Inspector which allows displaying the hierarchy of widgets that exist in the form. Following the Object Inspector is the property editor. Finally, in the center we may find the central Widget that can be a simple window or form, or a more complex Widget:

Once we add a Widget to the interface, a default name is assigned to the `objectName` field in the property editor. This name can be changed directly, as shown in figure 14.9. The same name will be the one used to reference the object from the Python code that handles the interface.

A natural way to see the result without embedding it in the final code is to use Qt Designer's preview mode, accessed by pressing **Ctrl + R**. In this mode the interface created is fully functional. Once the interface is complete, it must be saved. At that time a file with the `.ui` extension is created. Then it should be assembled with the Python code that controls the interface Widgets' functionalities. To integrate the interface with a given program, we use the `uic` module. This allows us to load the interface through the `(<interface-name>.ui)` method. The function returns a tuple with two elements: the first is a class named as the window defined in the `objectName` in *QtDesigner* Property

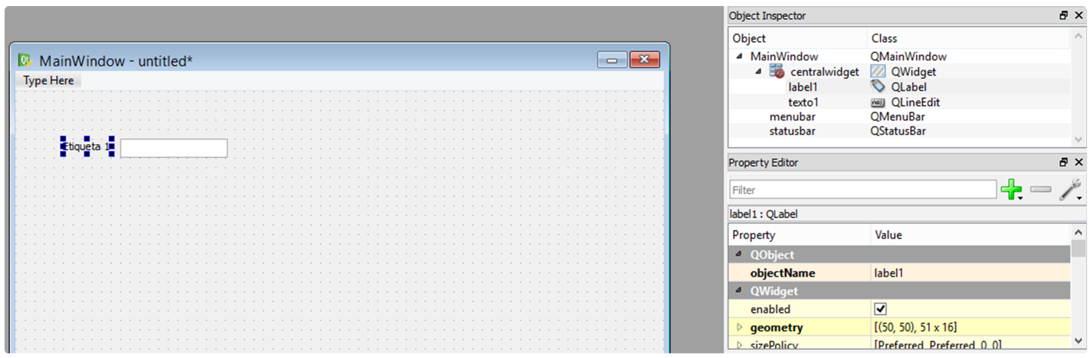


Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
horizontalLayout	QHBoxLayout
horizontalLayout_2	QHBoxLayout
horizontalSpacer	Spacer
horizontalSpacer_2	Spacer
label1	QLabel
pushButton	QPushButton
texto1	QLineEdit

MainWindow : QMainWindow

Property	Value
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [MS Shell Dlg 2, 8]

Figure 14.8: This figure shows the main panels available on *QtDesigner*. (a) Shows the Widget Box. (b) Shows the Object Inspector. (c) Shows the Property Editor.



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
label1	QLabel
texto1	QLineEdit
menubar	QMenuBar
statusbar	QStatusBar

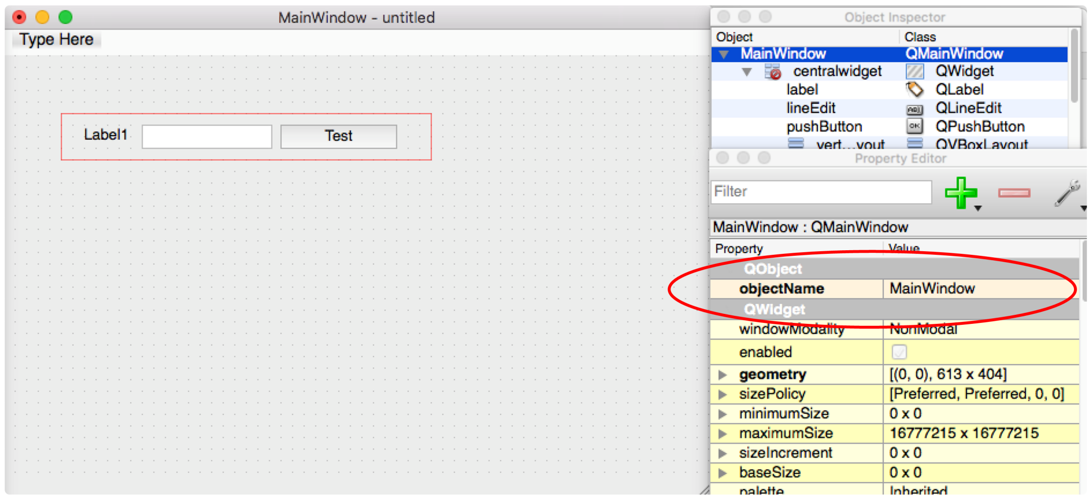
label1 : QLabel

Property	Value
QObject	
objectName	label1
QWidget	
enabled	✓
geometry	[(50, 50), 51 x 16]
sizePolicy	[Preferred, Preferred, 0, 0]

Figure 14.9: An example of how pyqt-designer assigns the object name to a new widget. In this case, the designer adds a new label call `label1`.

Editor; and the second is the name of the class from where it inherits. Figure 14.10 shows where the name appears in the `MainWindow` from the Property Editor.

The following example shows how to perform this procedure with an already created interface. In this case the form variable will include the tuple: `(class 'Ui_MainWindow' class 'PyQt4.QtGui.QMainWindow')`. The **Ui** prefix associated with the name of the class containing the interface is assigned by the `uic` module during interface loading. Once we loaded the interface, it must be initialized within the `__init__()` method, located in the class from where it inherits. This initialization is performed by the `setupUi(self)` method. Applications' creation



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
label	QLabel
lineEdit	QLineEdit
pushButton	QPushButton
vert...yout	QVBoxLayout

MainWindow : QMainWindow

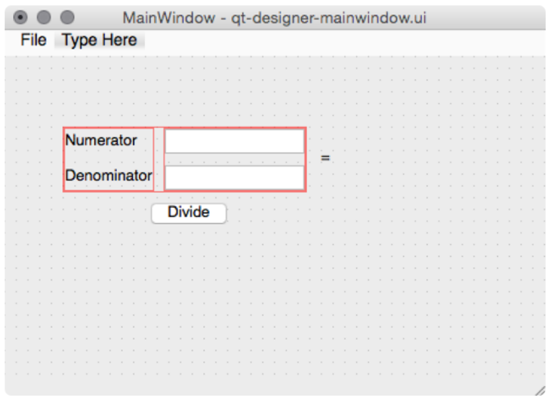
Property	Value
QObject	
objectName	MainWindow
QWidget	
windowModality	NonModal
enabled	☐
geometry	[(0, 0), 613 x 404]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited

Figure 14.10: We use a red ellipse to point out that the name that Qt Designer assigns to the class representing the window is, in this case `MainWindow`.

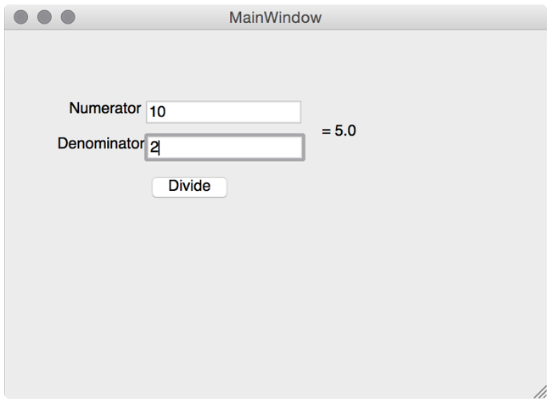
must be carried out by using the main program's structure, seen at the beginning of the explanation of graphical interfaces:

```python
# codes_11.py

from PyQt4 import QtGui, uic

form = uic.loadUiType("qt-designer-label.ui")


class MainWindow(form[0], form[1]):
    def __init__(self):
        super().__init__()
        self.setupUi(self)  # Interface is initialized


if __name__ == '__main__':
    # Application should be initialized just as if it had been
    # created manually
    app = QtGui.QApplication([])
    form = MainWindow()
    form.show()
    app.exec_()
```

Each Widget's action must be defined by signals and slots as explained before. Figure 14.11 shows an example of an interface that performs division between two numbers. The button created in *Qt Designer* displays the result on a label.



(a)



(b)

Figure 14.11: (a) The design view of the multiplication example. (b) The execution view.

```python
# codes_12.py

from PyQt4 import QtGui, uic

form = uic.loadUiType("qt-designer-mainwindow.ui")
print(form[0], form[1])


class MainWindow(form[0], form[1]):
    def __init__(self):
        super().__init__()
        # QtDesigner created interface is initialized
        self.setupUi(self)

        # Button signal is connected
        self.pushButton1.clicked.connect(self.divide)

    def divide(self):
        # This function acts as a slot for de button clicked signal
        self.label_3.setText('= ' + str(
            float(self.lineEdit1.text()) / float(
                self.lineEdit2.text())))
```

```


if __name__ == '__main__':
    app = QtGui.QApplication([])
    form = MainWindow()
    form.show()
    app.exec_()
```

It is easy to include new Widgets that simplify the user interaction within *Qt Designer*. One example is *radio buttons*, which allow us to capture user options on the form. Figure 14.12 shows a design form using radio buttons and the python code used to verify the values.

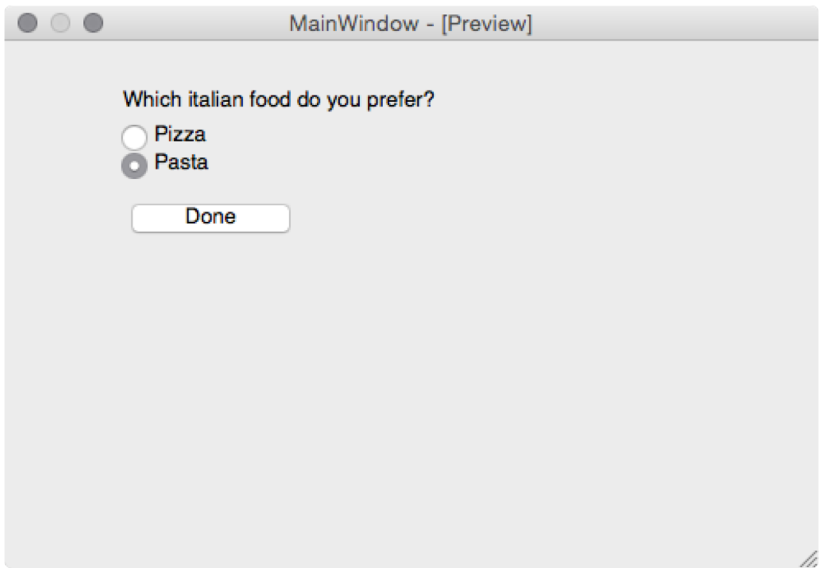


Figure 14.12: Figure shows an execution of a GUI using radio buttons.

```
# codes_13.py

from PyQt4 import QtGui, uic

form = uic.loadUiType("qt-designer-radiobutton.ui")
print(form[0], form[1])


class MainWindow(form[0], form[1]):
    def __init__(self):
        super().__init__()
        self.setupUi(self)

        self.pushButton1.clicked.connect(self.show_preferences)

    def show_preferences(self):
        for rb_id in range(1, 3):
            if getattr(self, 'radioButton' + str(rb_id)).isChecked():
                option = getattr(self, 'radioButton' + str(rb_id)).text()
                print(option)
                self.label2.setText('prefers: {0}'.format(option))
                self.label2.resize(self.label2.sizeHint())


if __name__ == '__main__':
    app = QtGui.QApplication([])
    form = MainWindow()
    form.show()
    app.exec_()
```