[Home](http://zetcode.com/) [Contents](http://zetcode.com/gui/pyqt5/) [Subscribe](http://zetcode.us13.list-manage.com/subscribe?u=9def9ccd4c70dbbaf691f90fc&id=6556210f80)

[Previous](http://zetcode.com/gui/pyqt5/layout/) [Next](http://zetcode.com/gui/pyqt5/dialogs/)

**Events and signals in PyQt5**

In this part of the PyQt5 programming tutorial, we will explore events and signals occurring in applications.

**Events**

GUI applications are event-driven. Events are generated mainly by the user of an application. But they can be generated by other means as well; e.g. an Internet connection, a window manager, or a timer. When we call the application's exec\_() method, the application enters the main loop. The main loop fetches events and sends them to the objects.

In the event model, there are three participants:

* event source
* event object
* event target

The *event source* is the object whose state changes. It generates events. The *event object* (event) encapsulates the state changes in the event source. The *event target* is the object that wants to be notified. Event source object delegates the task of handling an event to the event target.

PyQt5 has a unique signal and slot mechanism to deal with events. Signals and slots are used for communication between objects. A *signal* is emitted when a particular event occurs. A *slot* can be any Python callable. A slot is called when its connected signal is emitted.

**Signals and slots**

This is a simple example demonstrating signals and slots in PyQt5.

sigslot.py

#!/usr/bin/python3

# -\*- coding: utf-8 -\*-

"""

ZetCode PyQt5 tutorial

In this example, we connect a signal

of a QSlider to a slot of a QLCDNumber.

Author: Jan Bodnar

Website: zetcode.com

Last edited: January 2017

"""

import sys

from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import (QWidget, QLCDNumber, QSlider,

QVBoxLayout, QApplication)

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

lcd = QLCDNumber(self)

sld = QSlider(Qt.Horizontal, self)

vbox = QVBoxLayout()

vbox.addWidget(lcd)

vbox.addWidget(sld)

self.setLayout(vbox)

sld.valueChanged.connect(lcd.display)

self.setGeometry(300, 300, 250, 150)

self.setWindowTitle('Signal and slot')

self.show()

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

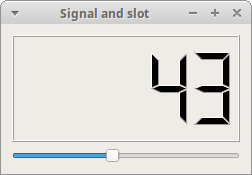
sys.exit(app.exec\_())

In our example, we display a QtGui.QLCDNumber and a QtGui.QSlider. We change the lcd number by dragging the slider knob.

sld.valueChanged.connect(lcd.display)

Here we connect a valueChanged signal of the slider to the display slot of the lcd number.

The *sender* is an object that sends a signal. The *receiver* is the object that receives the signal. The *slot* is the method that reacts to the signal.

Figure: Signal & slot

**Reimplementing event handler**

Events in PyQt5 are processed often by reimplementing event handlers.

escape.py

#!/usr/bin/python3

# -\*- coding: utf-8 -\*-

"""

ZetCode PyQt5 tutorial

In this example, we reimplement an

event handler.

Author: Jan Bodnar

Website: zetcode.com

Last edited: August 2017

"""

import sys

from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import QWidget, QApplication

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.setGeometry(300, 300, 250, 150)

self.setWindowTitle('Event handler')

self.show()

def keyPressEvent(self, e):

if e.key() == Qt.Key\_Escape:

self.close()

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

sys.exit(app.exec\_())

In our example, we reimplement the keyPressEvent() event handler.

def keyPressEvent(self, e):

if e.key() == Qt.Key\_Escape:

self.close()

If we click the Escape button, the application terminates.

**Event object**

Event object is a Python object that contains a number of attributes describing the event. Event object is specific to the generated event type.

eventobject.py

#!/usr/bin/python3

# -\*- coding: utf-8 -\*-

"""

ZetCode PyQt5 tutorial

In this example, we display the x and y

coordinates of a mouse pointer in a label widget.

Author: Jan Bodnar

Website: zetcode.com

Last edited: August 2017

"""

import sys

from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import QWidget, QApplication, QGridLayout, QLabel

class Example(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

grid = QGridLayout()

x = 0

y = 0

self.text = "x: {0}, y: {1}".format(x, y)

self.label = QLabel(self.text, self)

grid.addWidget(self.label, 0, 0, Qt.AlignTop)

self.setMouseTracking(True)

self.setLayout(grid)

self.setGeometry(300, 300, 350, 200)

self.setWindowTitle('Event object')

self.show()

def mouseMoveEvent(self, e):

x = e.x()

y = e.y()

text = "x: {0}, y: {1}".format(x, y)

self.label.setText(text)

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

sys.exit(app.exec\_())

In this example, we display the x and y coordinates of a mouse pointer in a label widget.

self.text = "x: {0}, y: {1}".format(x, y)

self.label = QLabel(self.text, self)

The x and y coordinates are displayd in a QLabel widget.

self.setMouseTracking(True)

Mouse tracking is disabled by default, so the widget only receives mouse move events when at least one mouse button is pressed while the mouse is being moved. If mouse tracking is enabled, the widget receives mouse move events even if no buttons are pressed.

def mouseMoveEvent(self, e):

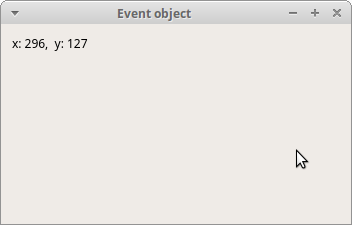
x = e.x()

y = e.y()

text = "x: {0}, y: {1}".format(x, y)

self.label.setText(text)

The e is the event object; it contains data about the event that was triggered; in our case, a mouse move event. With the x() and y() methods we determine the x and y coordinates of the mouse pointer. We build the string and set it to the label widget.

Figure: Event object

**Event sender**

Sometimes it is convenient to know which widget is the sender of a signal. For this, PyQt5 has the sender() method.

eventsource.py

#!/usr/bin/python3

# -\*- coding: utf-8 -\*-

"""

ZetCode PyQt5 tutorial

In this example, we determine the event sender

object.

Author: Jan Bodnar

Website: zetcode.com

Last edited: August 2017

"""

import sys

from PyQt5.QtWidgets import QMainWindow, QPushButton, QApplication

class Example(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

btn1 = QPushButton("Button 1", self)

btn1.move(30, 50)

btn2 = QPushButton("Button 2", self)

btn2.move(150, 50)

btn1.clicked.connect(self.buttonClicked)

btn2.clicked.connect(self.buttonClicked)

self.statusBar()

self.setGeometry(300, 300, 290, 150)

self.setWindowTitle('Event sender')

self.show()

def buttonClicked(self):

sender = self.sender()

self.statusBar().showMessage(sender.text() + ' was pressed')

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

sys.exit(app.exec\_())

We have two buttons in our example. In the buttonClicked() method we determine which button we have clicked by calling the sender() method.

btn1.clicked.connect(self.buttonClicked)

btn2.clicked.connect(self.buttonClicked)

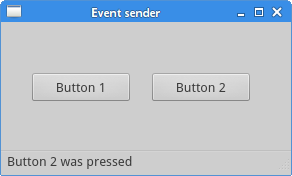
Both buttons are connected to the same slot.

def buttonClicked(self):

sender = self.sender()

self.statusBar().showMessage(sender.text() + ' was pressed')

We determine the signal source by calling the sender() method. In the statusbar of the application, we show the label of the button being pressed.

Figure: Event sender

**Emitting signals**

Objects created from a QObject can emit signals. The following example shows how we to emit custom signals.

customsignal.py

#!/usr/bin/python3

# -\*- coding: utf-8 -\*-

"""

ZetCode PyQt5 tutorial

In this example, we show how to

emit a custom signal.

Author: Jan Bodnar

Website: zetcode.com

Last edited: August 2017

"""

import sys

from PyQt5.QtCore import pyqtSignal, QObject

from PyQt5.QtWidgets import QMainWindow, QApplication

class Communicate(QObject):

closeApp = pyqtSignal()

class Example(QMainWindow):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.initUI()

def initUI(self):

self.c = Communicate()

self.c.closeApp.connect(self.close)

self.setGeometry(300, 300, 290, 150)

self.setWindowTitle('Emit signal')

self.show()

def mousePressEvent(self, event):

self.c.closeApp.emit()

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv)

ex = Example()

sys.exit(app.exec\_())

We create a new signal called closeApp. This signal is emitted during a mouse press event. The signal is connected to the close() slot of the QMainWindow.

class Communicate(QObject):

closeApp = pyqtSignal()

A signal is created with the pyqtSignal() as a class attribute of the external Communicate class.

self.c = Communicate()

self.c.closeApp.connect(self.close)

The custom closeApp signal is connected to the close() slot of the QMainWindow.

def mousePressEvent(self, event):

self.c.closeApp.emit()

When we click on the window with a mouse pointer, the closeApp signal is emitted. The application terminates.

In this part of the PyQt5 tutorial, we have covered signals and slots.