QGIS Plugins mit Python programmieren

Marco Bernasocchi Cédric Möri Stefan Ziegler

6. Mai 2011

1 Einführung

QGIS ist ein beliebtes Desktop GIS, das in C++ geschrieben ist. Neben der Möglichkeit, Erweiterungen in C++ zu programmieren, gibt es die Python Schnittstelle zu QGIS (kurz: PyQGIS). Es gibt drei Möglichkeiten, PyQGIS zu verwenden:

- Von der Pythonkonsole können in QGIS direkt Pythonanweisungen eingetippt werden. Das ist eher zu Testzwecken interessant.
- Es können Plugins in Python entwickelt werden, die dann ganz normal in QGIS mit dem Pluginmanager hinzugeladen werden können.
- Mit eigenen Python Anwendungen, welche auf die Core Bibliothek von QGIS zugreifen, aber eine eigene Benutzerschnittstelle mitbringen.

Dieser Workshop hat zum Ziel, einen einfachen Einstieg zur zweiten Möglichkeit, also dem Schreiben von Python Plugins, zu geben.

2 Wieso Python?

Python ist eine Skriptsprache, die mit dem Ziel entworfen wurde, einfach und übersichtlich zu sein. Ein weiterer Vorteil ist die grosse Verbreitung. Viele Programme, die zwar selber in C, C++ oder Java geschrieben sind, bieten die Möglichkeit, Erweiterungen in Python zu programmieren (z.B. OpenOffice oder Gimp). Ausserdem hat Python gegenüber C++ den Vorteil, eine automatische Speicherfreigabe (sog. garbage collector) zu haben, welche sich um nicht mehr benötigte Speicherbereiche kümmert.

3 Lizenz

PyQGIS Plugins verwenden Funktionalität von libqgis_core.so und libqgis_gui.so, welche beide unter der GPL lizensiert sind. Somit sind Plugins «derived work » im Sinne der GPL und müssen ebenfalls GPL sein. Sie können Ihre Plugins für jeden Zweck einsetzen und es besteht auch kein Zwang, sie zu veröffentlichen. Wenn Sie es aber tun, muss es unter den Bedingungen der GPL sein.

4 Was muss installiert werden, um loszulegen

Im Image oder auf dem USB-Stick ist alles nötige bereits installiert. Wenn Sie zu Hause PyQGIS Plugins programmieren möchten, brauchen Sie folgendes:

- QGIS
- Python
- Qt
- PyQT
- PyQt Entwicklungswerkzeuge

Wenn Sie Linux verwenden, gibt es für alle grösseren Distributionen Binärpakete. Verwenden Sie Windows, so enthält der PyQt Installer bereits Qt, PyQt und die PyQt Entwicklungswerkzeuge.

5 Ein PyQGIS Beispielplugin in fünf Schritten

Unser Beispielplugin ist bewusst einfach gehalten. Es fügt einen Knopf zur Menüleiste von QGIS hinzu. Wird der Knopf gedrückt, erscheint ein Dateidialog, mit dem man eine Shapedatei laden kann. Anschliessend erstellen wir einen eigenen Filedialog und verschieben die Geometrien der geladenen Datei um einen gewissen Betrag.

Für jedes Python Plugin muss ein eigener Ordner erzeugt werden, welcher die benötigten Dateien enthält. Am besten erzeugt man den Ordner in $HOME/.qgis/python/plugins (also in unserem Workshop /home/stefan/.qgis/python/plugins). Es besteht auch die Möglichkeit ein Plugin nach $QGIS_DIR/share/qgis/python/plugins zu kopieren resp. in diesem Verzeichnis zu erstellen.

5.1 Schritt 1: Das Plugin für den Pluginmanager bekanntmachen

Um das Plugin für den QGIS Pluginmanager verfügbar zu machen, muss es die Methoden `name()`, `description()` und `version()` implementieren, welche jeweils einen String mit der gewünschten Information zurückgeben. Ausserdem bietet es die Methode `classFactory(QgisInterface)` an, die es dem Pluginmanager von QGIS erlaubt, eine Instanz des Plugins zu erzeugen. Das Objekt `QgisInterface`, welches mitgegeben wird, erlaubt es dem Plugin, auf gewisse Funktionen von QGIS zuzugreifen. In unserem Beispielplugin brauchen wir dieses Objekt aber erst ab Schritt 2.

Beachten Sie, dass in Python, im Unterschied zu anderen Programmiersprachen, die Einrückung sehr wichtig ist. Der Pythoninterpreter gibt eine Fehlermeldung aus, wenn hier etwas nicht stimmt.

Im Verzeichnis $HOME/.qgis/python/plugins/workshopplugin erzeugen wir nun zwei neue Dateien:

- Die Datei *workshop.py* enthält die eigentliche Pluginklasse:

```
# -*- coding: utf-8 -*-

from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *

class Workshop:

  def __init__(self,iface):
    #Referenz zum Qgis-Interface sichern
    self.iface = iface

  def initGui(self):
    print "Initialisiere das GUI"

  def unload(self):
    print "Entlade das Plugin"
```

- Die Datei *__init__.py* enthält die Methoden `name()`, `description()`, `version()` und `classFactory()`. Da mit der `classFactory()` Methode eine neue Instanz unserer Pluginklasse angelegt werden kann, muss der Code dieser Klasse importiert werden:

```
# -*- coding: utf-8 -*-
from workshop import Workshop

def name():
  return "Workshop Plugin"

def description():
  return "Ein einfaches Beispielplugin"

def version():
  return "Version 0.1"

def qgisMinimumVersion():
  return "1.6"

def authorName():
  return "Max Muster"

def icon():
  return "icon1.png"

def classFactory(iface):
  return Workshop(iface)
```

Nun sollte das Plugin mit dem Namen «Workshop Plugin» im Pluginmanager erscheinen und geladen bzw. entladen werden können.

5.2 Schritt 2: Button und Menü hinzufügen

5.2.1 Icon

Um das Button-Icon für unser Programm verfügbar zu machen, generieren wir ein sogenanntes resource file. Darin ist dann der Inhalt der Graphik in hexadezimalem Format enthalten. Darum brauchen wir uns aber nicht zu kümmern, sondern wir verwenden den ***pyrcc*** Compiler, der das, ausgehend von der Datei *resources.qrc*, für uns macht. Der Inhalt der Datei sollte wie folgt aussehen:

```
<RCC>
    <qresource prefix="/plugins/workshopplugin">
        <file>icon1.png</file>
    </qresource>
</RCC>
```

Öffnen Sie nun eine Shell, wechseln ins Pluginverzeichnis und geben Sie ein: `pyrcc4 -o resources.py resources.qrc`.

5.2.2 Menü und Knopf hinzufügen

Dann können wir in unserem Programm die Methode `initGui()` und `unload()` implementieren. Wir erzeugen ein `QAction` Objekt, welches bei Aktivierung die `run()` Methode des Plugins ausführt. Damit können wir den Menüeintrag und den Knopf generieren:

```
# -*- coding: utf-8 -*-

from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *

import resources

class Workshop:

  def __init__(self,iface):
    #Referenz zum Qgis-Interface sichern
    self.iface = iface

  def initGui(self):
    #Action um die Pluginkonfiguration zu starten
    self.action = QAction(QIcon(":/plugins/workshopplugin/icon1.png"),"Workshop-Plugin",
        self.iface.mainWindow())

    #Action mit der run-Methode verknüpfen
    QObject.connect(self.action, SIGNAL("activated()"), self.run)

    #Toolbar-Button und Menu-Eintrag generieren
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("Workshop Plugin", self.action)

```

```
26
27    def unload(self):
28      #Toolbar-Button und Menu-Eintrag wieder entfernen
29      self.iface.removePluginMenu("Workshop Plugin", self.action)
30      self.iface.removeToolBarIcon(self.action)
31
32    def run(self):
33      QMessageBox.information(None, "Gruss", "Hallo Welt!")
```

5.3 Schritt 3: Shapefile laden

Jetzt implementieren wir die eigentliche Funktionalität des Plugins in der Methode `run()`. Wir rufen die Methode `QFileDialog.getOpenFileName()` auf, die einen Dateidialog anzeigt und uns den Pfad der gewählten Datei liefert. Falls der Benutzer den Dialog abgebrochen hat, ist der Pfad Null. Wir rufen die Methode `addVectorLayer()` des Interfaceobjekts auf, wodurch die gewünschte Ebene geladen wird. Die Methode braucht drei Argumente: den Pfad, einen Namen für den Layer und eine Providerbezeichnung. Bei Shapedateien ist dies «ogr», weil QGIS intern die OGR Bibliothek verwendet, um auf Shapedateien zuzugreifen:

```
1  # -*- coding: utf-8 -*-
2
3  from PyQt4.QtCore import *
4  from PyQt4.QtGui import *
5  from qgis.core import *
6
7  import resources
8
9  class Workshop:
10
11   def __init__(self,iface):
12     #Referenz zum Qgis-Interface sichern
13     self.iface = iface
14
15   def initGui(self):
16     #Action um die Pluginkonfiguration zu starten
17     self.action = QAction(QIcon(":/plugins/workshopplugin/icon1.png"),"Workshop-Plugin",
           self.iface.mainWindow())
18
19     #Action mit der run-Methode verknüpfen
20     QObject.connect(self.action, SIGNAL("activated()"), self.run)
21
22     #Toolbar-Button und Menu-Eintrag generieren
23     self.iface.addToolBarIcon(self.action)
24     self.iface.addPluginToMenu("Workshop Plugin", self.action)
25
26
27   def unload(self):
28     #Toolbar-Button und Menu-Eintrag wieder entfernen
29     self.iface.removePluginMenu("Workshop Plugin", self.action)
30     self.iface.removeToolBarIcon(self.action)
```

```
31
32    def run(self):
33      fileName = QFileDialog.getOpenFileName(None, QString.fromLocal8Bit("Wählen Sie eine
            Datei aus"), "", "*.shp *.gml")
34      if fileName.isNull():
35        QMessageBox.information(None, "Abbruch", QString.fromLocal8Bit("Keine Datei
              ausgewählt!"))
36      else:
37        vlayer = self.iface.addVectorLayer(fileName, "mein Layer", "ogr")
```

5.4 Schritt 4: Eigenen Filedialog entwerfen

Als nächstes erstellen wir einen eigenen Filedialog, der uns zudem erlaubt den Namen des geladenen Layers in der Legende umzubenennen. Dazu verwenden wir den Qt Designer. Der Qt Designer ist ein Programm zum Erstellen von sogenannten Grafischen User Interfaces (GUI). Für unsere Aufgabe brauchen wir sogenannte «LineEdits», um den Pfad der zu ladenden Datei und den gewünschen neuen Namen in der Legende anzuzeigen. Zusätzlich brauchen wir mehrere Buttons, die verschiedene Aktionen ausführen.

Der soeben erstellte eigene Dialog kann unter dem Namen «layerladen.ui» im Plugin Verzeichnis gespeichert werden. Der Qt Designer speichert diese Dialoge als XML-Datei ab. Damit kann QGIS aber nichts anfangen und wir müssen die Datei mit dem Programm ***pyuic*** in Python Code umschreiben lassen. Öffnen Sie nun eine Shell, wechseln ins Pluginverzeichnis und geben Sie ein: `pyuic4 -o ui_layerladen.py layerladen.ui`. Die Datei «ui_layerladen.py» sollte wie folgt aussehen:

```
1  # -*- coding: utf-8 -*-
2
3  # Form implementation generated from reading ui file 'layerladen.ui'
4  #
5  # Created: Thu Apr 28 17:25:46 2011
6  #      by: PyQt4 UI code generator 4.5.4
7  #
8  # WARNING! All changes made in this file will be lost!
9
10 from PyQt4 import QtCore, QtGui
11
12 class Ui_LayerladenDialog(object):
13     def setupUi(self, LayerladenDialog):
14         LayerladenDialog.setObjectName("LayerladenDialog")
15         LayerladenDialog.resize(442, 203)
16         self.gridLayout_4 = QtGui.QGridLayout(LayerladenDialog)
17         self.gridLayout_4.setObjectName("gridLayout_4")
18         self.formLayout = QtGui.QFormLayout()
19         self.formLayout.setObjectName("formLayout")
20         self.gridLayout_3 = QtGui.QGridLayout()
21         self.gridLayout_3.setObjectName("gridLayout_3")
22         self.gridLayout = QtGui.QGridLayout()
23         self.gridLayout.setObjectName("gridLayout")
24         self.le_filename = QtGui.QLineEdit(LayerladenDialog)
```

```
25          self.le_filename.setObjectName("le_filename")
26          self.gridLayout.addWidget(self.le_filename, 0, 0, 1, 1)
27          self.pb_layerdialog = QtGui.QPushButton(LayerladenDialog)
28          self.pb_layerdialog.setObjectName("pb_layerdialog")
29          self.gridLayout.addWidget(self.pb_layerdialog, 1, 0, 1, 1)
30          self.gridLayout_3.addLayout(self.gridLayout, 0, 0, 1, 1)
31          self.gridLayout_2 = QtGui.QGridLayout()
32          self.gridLayout_2.setObjectName("gridLayout_2")
33          self.label_2 = QtGui.QLabel(LayerladenDialog)
34          self.label_2.setObjectName("label_2")
35          self.gridLayout_2.addWidget(self.label_2, 0, 0, 1, 1)
36          self.le_layername = QtGui.QLineEdit(LayerladenDialog)
37          self.le_layername.setObjectName("le_layername")
38          self.gridLayout_2.addWidget(self.le_layername, 1, 0, 1, 1)
39          self.gridLayout_3.addLayout(self.gridLayout_2, 1, 0, 1, 1)
40          self.formLayout.setLayout(0, QtGui.QFormLayout.FieldRole, self.gridLayout_3)
41          self.gridLayout_4.addLayout(self.formLayout, 0, 0, 1, 1)
42          self.buttonBox = QtGui.QDialogButtonBox(LayerladenDialog)
43          self.buttonBox.setOrientation(QtCore.Qt.Horizontal)
44          self.buttonBox.setStandardButtons(QtGui.QDialogButtonBox.Cancel|QtGui.
                QDialogButtonBox.Ok)
45          self.buttonBox.setObjectName("buttonBox")
46          self.gridLayout_4.addWidget(self.buttonBox, 1, 0, 1, 1)
47
48          self.retranslateUi(LayerladenDialog)
49          QtCore.QObject.connect(self.buttonBox, QtCore.SIGNAL("accepted()"),
                LayerladenDialog.accept)
50          QtCore.QObject.connect(self.buttonBox, QtCore.SIGNAL("rejected()"),
                LayerladenDialog.reject)
51          QtCore.QMetaObject.connectSlotsByName(LayerladenDialog)
52
53      def retranslateUi(self, LayerladenDialog):
54          LayerladenDialog.setWindowTitle(QtGui.QApplication.translate("LayerladenDialog", "
                Dialog", None, QtGui.QApplication.UnicodeUTF8))
55          self.pb_layerdialog.setText(QtGui.QApplication.translate("LayerladenDialog", "Layer
                 auswählen", None, QtGui.QApplication.UnicodeUTF8))
56          self.label_2.setText(QtGui.QApplication.translate("LayerladenDialog", "Name des
                Layers, der dargestellt werden soll:", None, QtGui.QApplication.UnicodeUTF8))
```

Um die vorher erstellten Buttons mit Leben zu füllen, müssen wir die Datei «layerladenGui.py» mit den beiden Methoden `on_pb_layerdialog_clicked()` und `accept()` erstellen:

```
1  # -*- coding: utf-8 -*-
2
3  from PyQt4.QtCore import *
4  from PyQt4.QtGui import *
5  from qgis.core import *
6  from ui_layerladen import Ui_LayerladenDialog
7  import os, sys
8
9  class LayerLadenGui(QDialog, Ui_LayerladenDialog):
10
11   def __init__(self, parent):
```

```
    QDialog.__init__(self, parent)
    self.setupUi(self)

  @pyqtSignature("on_pb_layerdialog_clicked()")
  def on_pb_layerdialog_clicked(self):
    fileName = QFileDialog.getOpenFileName(None, QString.fromLocal8Bit("Wählen Sie eine
        Datei aus"), "", "*.shp *.gml")
    if fileName.isNull():
      QMessageBox.information(None, "Abbruch", QString.fromLocal8Bit("Keine Datei
          ausgewählt!"))
    else:
      self.le_filename.setText(fileName)



  def accept(self):
      self.emit(SIGNAL("layername( PyQt_PyObject )"), self.le_layername.text())
      self.emit(SIGNAL("filename( PyQt_PyObject )"), self.le_filename.text())
      self.close()
```

In der bestehenden Datei«workshop.py» muss die Methode `run()` angepasst werden und zwei neue Methoden (`setLayerName()` und `layerladen()`) definiert werden:

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

from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *

from layerladenGui import *

import resources

class Workshop:

  def __init__(self,iface):
    #Referenz zum Qgis-Interface sichern
    self.iface = iface

  def initGui(self):
    #Action um die Pluginkonfiguration zu starten
    self.action = QAction(QIcon(":/plugins/workshopplugin/icon1.png"),"Workshop-Plugin",
        self.iface.mainWindow())

    #Action mit der run-Methode verknüpfen
    QObject.connect(self.action, SIGNAL("activated()"), self.run)

    #Toolbar-Button und Menu-Eintrag generieren
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("Workshop Plugin", self.action)


  def unload(self):
    #Toolbar-Button und Menu-Eintrag wieder entfernen
```

```
31      self.iface.removePluginMenu("Workshop Plugin", self.action)
32      self.iface.removeToolBarIcon(self.action)
33
34    def run(self):
35      self.ctrl = LayerLadenGui(self.iface.mainWindow())
36      QObject.connect(self.ctrl, SIGNAL("layername( PyQt_PyObject )"), self.setLayerName)
37      QObject.connect(self.ctrl, SIGNAL("filename( PyQt_PyObject )"), self.layerladen)
38      self.ctrl.show()
39
40    def setLayerName(self, layerName):
41      self.layerName = layerName
42
43
44    def layerladen(self, fileName):
45       vlayer = self.iface.addVectorLayer(fileName, self.layerName, "ogr")
```

5.5 Schritt 5: Feature verschieben

Als letzter Schritt implementieren wir eine Translation, welche die geladenen Feature um einen gewissen Betrag verschiebt. Dazu brauchen wir einen zweiten Button. Für das Icon müssen wir die *resources.qrc* Datei anpassen und nochmals den ***pyrcc*** Befehl ausführen (siehe Schritt 2). Anschliessend passen müssen wir in der Datei «workshop.py» verschiedene Methoden anpassen:

```
1   # -*- coding: utf-8 -*-
2
3   from PyQt4.QtCore import *
4   from PyQt4.QtGui import *
5   from qgis.core import *
6
7   from layerladenGui import *
8
9   import resources
10
11  class Workshop:
12
13    def __init__(self,iface):
14      #Referenz zum Qgis-Interface sichern
15      self.iface = iface
16
17    def initGui(self):
18      #Action um die Pluginkonfiguration zu starten
19      self.action = QAction(QIcon(":/plugins/workshopplugin/icon1.png"),"Workshop-Plugin
            Laden",  self.iface.mainWindow())
20      #Action mit der run-Methode verknüpfen
21      QObject.connect(self.action, SIGNAL("activated()"), self.run)
22
23      #Action für den Move eines features
24      self.actionMove = QAction(QIcon(":/plugins/workshopplugin/icon2.png"),"Workshop-Plugin
            Move",  self.iface.mainWindow())
25      #Action mit der run-Methode verknÃ¼pfen
26      QObject.connect(self.actionMove, SIGNAL("activated()"), self.runMove)
```

```
27
28      #Toolbar-Button und Menu-Eintrag generieren
29      self.iface.addToolBarIcon(self.action)
30      self.iface.addToolBarIcon(self.actionMove)
31      self.iface.addPluginToMenu("Workshop Plugin", self.action)
32      self.iface.addPluginToMenu("Workshop Plugin", self.actionMove)
33
34
35    def unload(self):
36      #Toolbar-Button und Menu-Eintrag wieder entfernen
37      self.iface.removePluginMenu("Workshop Plugin", self.action)
38      self.iface.removePluginMenu("Workshop Plugin", self.actionMove)
39      self.iface.removeToolBarIcon(self.action)
40      self.iface.removeToolBarIcon(self.actionMove)
41
42    def run(self):
43      self.ctrl = LayerLadenGui(self.iface.mainWindow())
44      QObject.connect(self.ctrl, SIGNAL("layername( PyQt_PyObject )"), self.setLayerName)
45      QObject.connect(self.ctrl, SIGNAL("filename( PyQt_PyObject )"), self.layerladen)
46      self.ctrl.show()
47
48    def setLayerName(self, layerName):
49      self.layerName = layerName
50
51
52    def layerladen(self, fileName):
53       vlayer = self.iface.addVectorLayer(fileName, self.layerName, "ogr")
54
55
56    def runMove(self):
57      mc = self.iface.mapCanvas()
58
59      layer = mc.currentLayer()
60      if layer <> None:
61        if(layer.type()==0):
62          layer.startEditing()
63
64          if layer.isEditable():
65            features = layer.selectedFeaturesIds()
66
67            for f in features:
68              layer.translateFeature(f, 100, 100)
69
70            layer.commitChanges()
71
72            mc.refresh()
```

6 Weitere Informationen

Wie Sie sehen, braucht es einige Bausteine, um PyQGIS Plugins schreiben zu können. Man muss Python und die QGIS Pluginschnittstelle kennen sowie die Qt Klassen und Tools verwenden. Das beste Vorgehen am Anfang ist, anhand von Beispielen zu lernen

und die Mechanismen bereits existierender Plugins zu kopieren. Mit dem Plugin Installer, der übrigens selber ein Python Plugin ist, können Sie sich eine Vielzahl solcher Erweiterungen auf den Rechner laden und ihre Funktionsweise studieren.

Zu den einzelnen Themen gibt es auch ausführliche Online Referenzen:

- QGIS wiki: http://wiki.qgis.org/qgiswiki/PythonBindings
- QGIS API Dokumentation: http://doc.qgis.org/index.html
- Qt Dokumentation: http://doc.trolltech.com/4.6/index.html
- PyQt: http://www.riverbankcomputing.co.uk/pyqt/
- Python tutorial: http://docs.python.org/

Dieses Tutorial basiert auf Text und Codebeispielen von Dr. Horst Düster, Dr. Marco Hugentobler und Otto Dassau anlässlich der FOSSGIS 2009.