# Do it yourself: Wie kann ich mit Python ein QGIS-Plugin programmieren?


**Carmen Viesca Álvarez**

GIS-Consultant

carmen.viesca@wheregroup.com

**Isabelle Korsch**

GIS-Consultant

isabelle.korsch@wheregroup.com

# Agenda

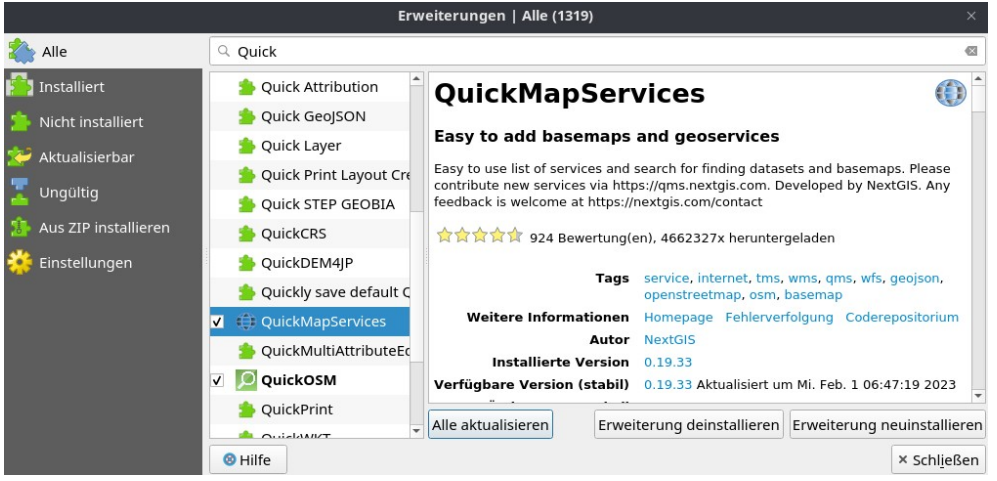
01. QGIS-Plugins: Einführung
02. Python-Grundlagen
03. PyQGIS: Suchfunktion programmieren
04. Plugin-Grundstruktur
05. Plugin: Suchfunktion integrieren
06. Tipps, Tricks & Links

# QGIS-Plugins

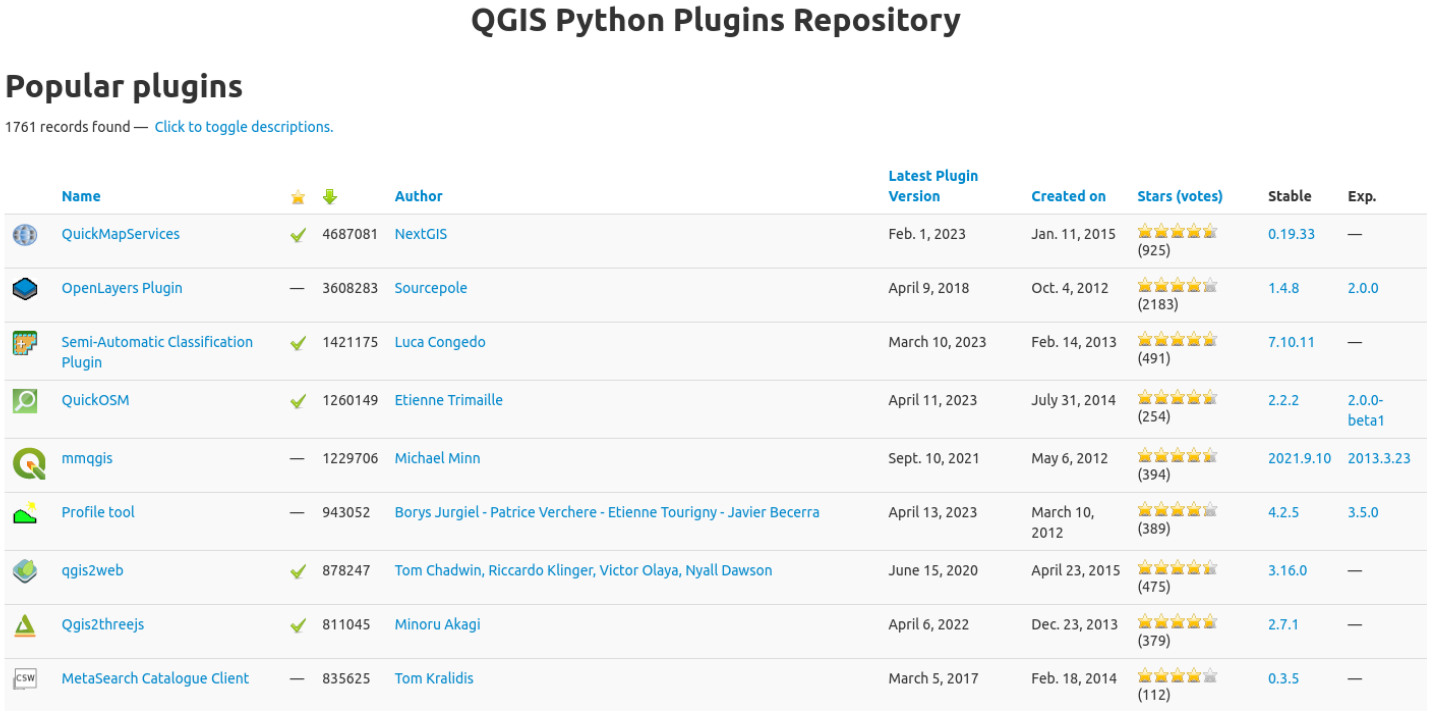
Erweiterungsmanager:

Erweiterungen → Erweiterungen verwalten und installieren

- Installation und Aktualisierung von Plugins
- Offizielles QGIS-Repositorium
- Kernplugins
- Weitere Plugins
- Unternehmensinterne Repositorien
- Plugins werden im Profil gespeichert



# QGIS-Plugins



QGIS Python Plugins Repository

Popular plugins

1761 records found — Click to toggle descriptions.

| Name | | Author | Latest Plugin Version | Created on | Stars (votes) | Stable | Exp. |
|---|---|---|---|---|---|---|---|
| QuickMapServices | ✓ 4687081 | NextGIS | Feb. 1, 2023 | Jan. 11, 2015 | (925) | 0.19.33 | — |
| OpenLayers Plugin | — 3608283 | Sourcepole | April 9, 2018 | Oct. 4, 2012 | (2183) | 1.4.8 | 2.0.0 |
| Semi-Automatic Classification Plugin | ✓ 1421175 | Luca Congedo | March 10, 2023 | Feb. 14, 2013 | (491) | 7.10.11 | — |
| QuickOSM | ✓ 1260149 | Etienne Trimaille | April 11, 2023 | July 31, 2014 | (254) | 2.2.2 | 2.0.0-beta1 |
| mmqgis | — 1229706 | Michael Minn | Sept. 10, 2021 | May 6, 2012 | (394) | 2021.9.10 | 2013.3.23 |
| Profile tool | — 943052 | Borys Jurgiel - Patrice Verchere - Etienne Tourigny - Javier Becerra | April 13, 2023 | March 10, 2012 | (389) | 4.2.5 | 3.5.0 |
| qgis2web | ✓ 878247 | Tom Chadwin, Riccardo Klinger, Victor Olaya, Nyall Dawson | June 15, 2020 | April 23, 2015 | (475) | 3.16.0 | — |
| Qgis2threejs | ✓ 811045 | Minoru Akagi | April 6, 2022 | Dec. 23, 2013 | (379) | 2.7.1 | — |
| MetaSearch Catalogue Client | — 835625 | Tom Kralidis | March 5, 2017 | Feb. 18, 2014 | (112) | 0.3.5 | — |

https://plugins.qgis.org/plugins/popular/

# C++ oder Python?

- QGIS wurde in C++ programmiert (https://api.qgs.org/api)
- Python Schnittstelle: PyQGIS: Python Binding der C++ Bibliothek (https://qgis.org/pyqgis/master)
- Python ist eine Skriptsprache und „gut geeignet“ als Schnittstellensprache


# Wofür können wir die Python-Schnittstelle nutzen?

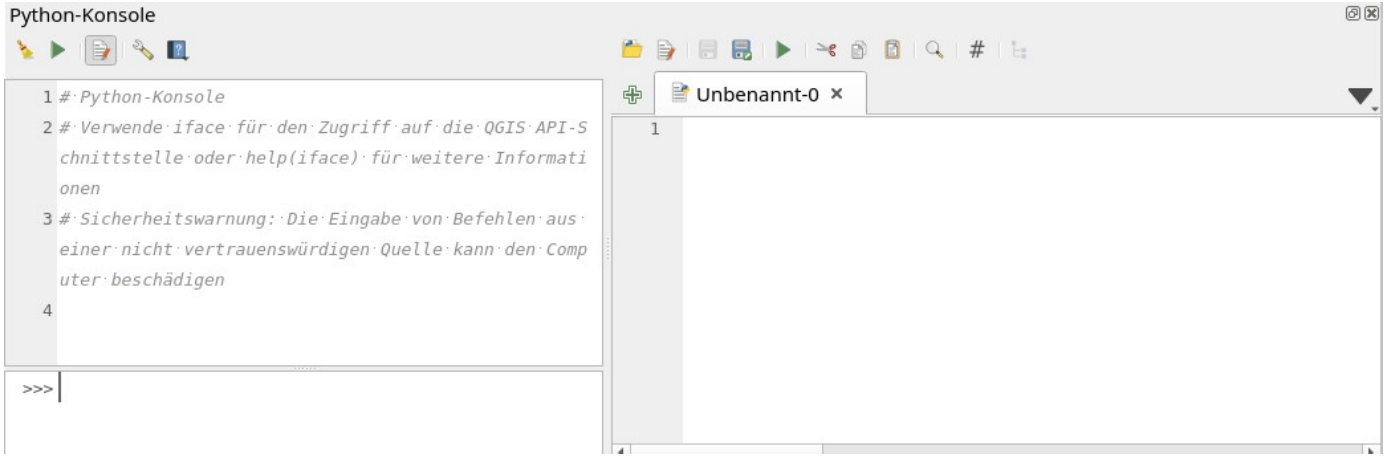
- Einsatz in der täglichen Arbeit mit QGIS:
  - Integrierte Python Konsole: Kurze Befehle oder Skripte
  - Ausdrucksеditor: eigene Funktionen erstellen
  - Python Code beim Start von QGIS ausführen
  - etc.
- Oder… neue Funktionen schreiben:
  - **Plugins schreiben!**
  - Neue Werkzeuge erstellen
  - Standalone Applications

# Warum Python?

- Open Source
- Lesbarkeit und Kürze
- Skriptsprache
- Gute Sprache für den Einstieg ins Programmieren
- Trotzdem: mächtig, vielfältige Einsatzbereiche, leicht erweiterbar (https://pypi.org)
- Kostenlos
- etc.

# Python-Konsole in QGIS

- Python 3 ist mit QGIS automatisch installiert
- Konsole öffnen: Erweiterungen → Python-Konsole
- Konsole für kurze Befehle
- Editor für Python-Skripte



# Python-Grundlagen

→ Bitte öffnen Sie das Skript *python_basics.py*

# PyQGIS-Grundlagen

→ bitte öffnen Sie das Skript *pyqgis_beispiele.py*

# Suchfunktion programmieren

```python
# iface -> Zugriff auf QGIS-Oberflaeche
# Aktiven Layer in lyr abspeichern
lyr = iface.activeLayer()

# Einzelne features aus dem Layer abspeichern
feats = lyr.getFeatures()

# Einzelne Features durchlaufen
for feat in feats:
    # Attribute des Features berechnen
    attrs = feat.attributes()
    # Attribute in die Konsole schreiben
    print(attrs)
```

- iface.activeLayer() → gerade aktivierter Layer
- getFeatures() → Erstellt eine Liste der Features aus dem Layer
- for feat in feats → Die Liste der Features wird durchlaufen
- feat.attributes() → Attributwerte des jeweiligen Features berechnen

# Suchfunktion programmieren

```python
searchtext = 'Maison'

# iface -> Zugriff auf QGIS-Oberflaeche
# Aktiven Layer in lyr abspeichern
lyr = iface.activeLayer()

# Einzelne features aus dem Layer abspeichern
feats = lyr.getFeatures()

# Einzelne Features durchlaufen
for feat in feats:
    attrs = feat.attributes()
    # Suchtext verwenden
    if searchtext in str(attrs):
        print(attrs)
```

- Suchtext verwenden
- Test: Ist der Suchtext in einem Attributwert enthalten?

# Suchfunktion programmieren

```python
searchtext = 'Maison'

# iface -> Zugriff auf QGIS-Oberflaeche
# Aktiven Layer in lyr abspeichern
lyr = iface.activeLayer()

# Einzelne features aus dem Layer abspeichern
feats = lyr.getFeatures()

# Einzelne Features durchlaufen
for feat in feats:
    attrs = feat.attributes()

    # Suchtext verwenden
    if searchtext in str(attrs):

        # Zum gefundenen Feature zoomen
        # Feature ID berechnen
        fid = feat.id()
        # Feature selektieren
        lyr.selectByIds([fid])
        # Zum Feature zoomen
        iface.actionZoomToSelected().trigger()
        break
```

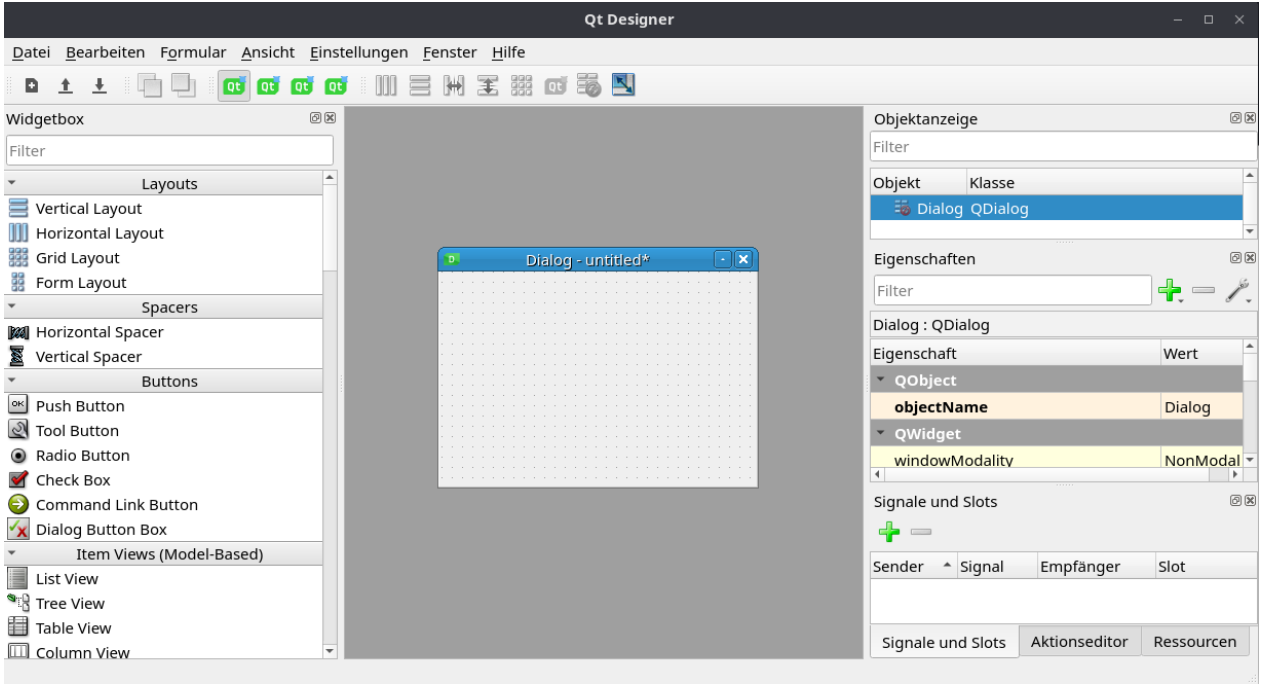
Beim ersten Feature, das gefunden wird:

- Feature-ID berechnen
- Feature im aktiven Layer selektieren
- iface.actionZoomToSelected().trigger() → Zum selektierten Feature zoomen

# PyQt

- Python Binding von der C++ Bibliothek Qt
- Entwickelt von der Firma Riverbank Computing
- Bibliothek zum Erstellen von graphischen Benutzeroberflächen (GUIs)
- Sehr beliebt: Spotify, Google Earth, OBS ...
- … und alle Oberflächen in QGIS wurden mit Qt erstellt
- Dokumentation: https://doc.qt.io/qtforpython-5/#documentation

# PyQt - Qt Designer




- Programm zum Erstellen von Oberflächen
- Reduziert den Programmieraufwand → Nur die Interaktion mit der GUI muss programmiert werden
- Häufig direkt mit QGIS installiert (z.Bsp QGIS Netzwerkinstallation)

Manual:

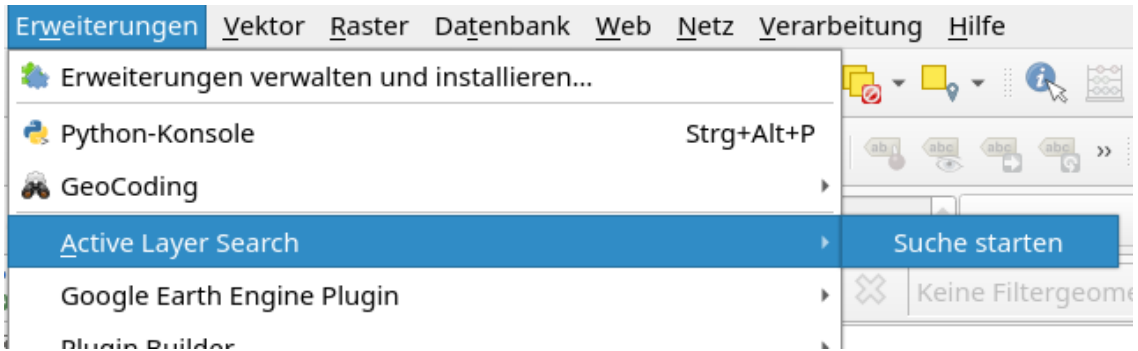
https://doc.qt.io/qt-5/qtdesigner-manual.html

# Plugin-Grundstruktur

- Wir verwenden nun ein Template, um die Plugin-Grundstruktur kennenzulernen
- Bitte installieren Sie *active_layer_search.zip* über den Erweiterungsmanager

# Platzierung des Plugins

- Das Plugin erscheint unter Erweiterungen
- Wir werden später sehen, wie das Plugin mit einem Icon platziert werden kann




# Plugin-Grundstruktur

- Was passiert, wenn ein Plugin „installiert“ wird?

  → Der Ordner wird entpackt und im verwendeten QGIS-Profil abgelegt
- QGIS: Einstellungen → Benutzerprofile → Aktuelles Profilverzeichnis öffnen
- Im Profilverzeichnis: python → plugins
- Deinstallation eines Plugins: Der Ordner wird aus dem plugins-Ordner wieder entfernt
- Alternative zur .zip Datei: Direkt im plugins-Ordner das Plugin ablegen

# Plugin-Grundstruktur

**Was muss in einem Plugin enthalten sein?**

- *metadata.txt*

  → Informationen über das Plugin

- *__init__.py*

  → Einstiegsstelle für QGIS, um das Plugin zu laden

- *active_layer_search.py*

  → enthält das eigentliche Plugin/ sorgt dafür, dass das Plugin in der QGIS-Oberfläche erscheint

# PyQGIS-Grundlagen

**Was kann in einem Plugin enthalten sein?**

- *search_dialog.ui*

  → Format, in dem eine graphische Benutzeroberfläche abgespeichert werden kann (zum Beispiel erstellt mit dem Qt Designer)

- *icon.png*

  → Icon fürs Menü

- Und noch vieles mehr (nutzen wir heute nicht):

  Noch mehr Python Dateien, Bilder, Readme-Datein...

# Plugin-Grundstruktur

| Name | Funktion | Verpflichtend? | Name fest? |
|---|---|---|---|
| metadata.txt | Metainformationen über das Plugin | ja | ja |
| __init__.py | Einstiegspunkt für QGIS | ja | ja |
| active_layer_search.py | Enthält das eigentliche Plugin | ja | nein |
| search_dialog.ui | Enthält die graphische Benutzeroberfläche | nein | nein |

# Wo schreibt man ein Plugin?

- Das Plugin wird **nicht von uns** in der Python-Konsole in QGIS ausgeführt
- QGIS führt die Plugin-Dateien aus, wenn wir das Plugin in QGIS verwenden
- Wir können das Plugin in einem beliebigen Editor erstellen
- Natürlich können wir auch eine Python IDE verwenden, aber:
  - Nur für Syntax-Hilfen, Autovervollständigung etc.
  - Der Code wird **nicht** in der IDE ausgeführt
- Wir verwenden in diesem Workshop den Editor **Geany**

# metadata.txt

metadata.txt

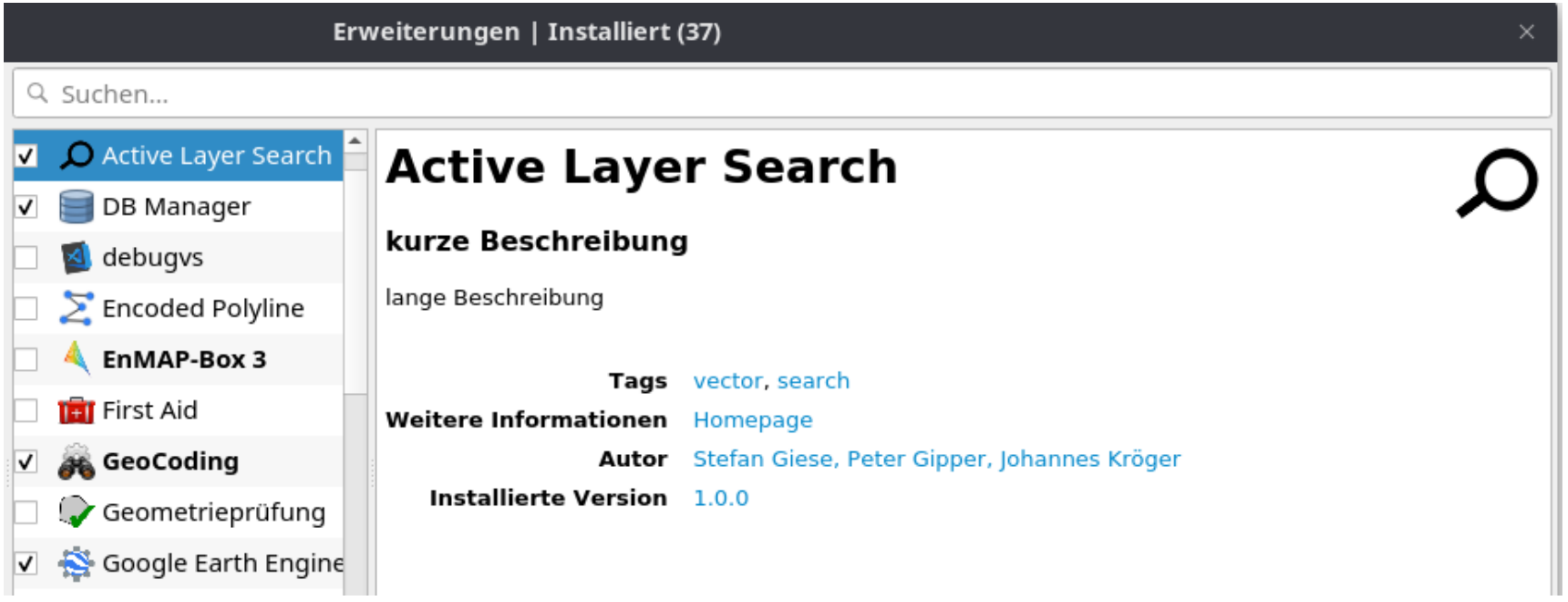
```
[general]
name=Active Layer Search
author=Stefan Giese, Peter Gipper, Johannes Kröger
email=peter.gipper@wheregroup.com
description=kurze Beschreibung
about=lange Beschreibung
homepage=https://foss-academy.com/
tags=vector, search

version=1.0.0
qgisMinimumVersion=3.0
qgisMaximumVersion=3.99

icon=icon.svg
experimental=False
deprecated=False

```

- Metadaten zum Plugin
- Obligatorische und einige optionale Angaben
- Dokumentation unter:

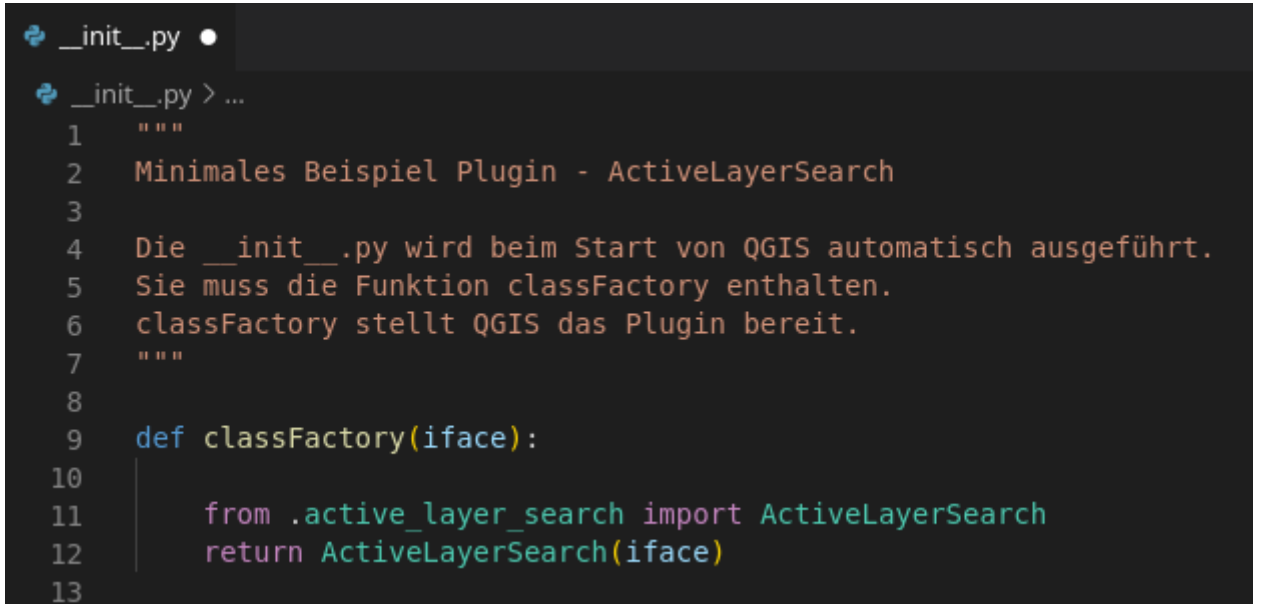
  https://docs.qgis.org/testing/en/docs/pyqgis_developer_cookbook/plugins/plugins.html#plugin-metadata

# metadata.txt

Die Metadaten erscheinen im Erweiterungsmanager:



# __init__.py




- Erinnerung:

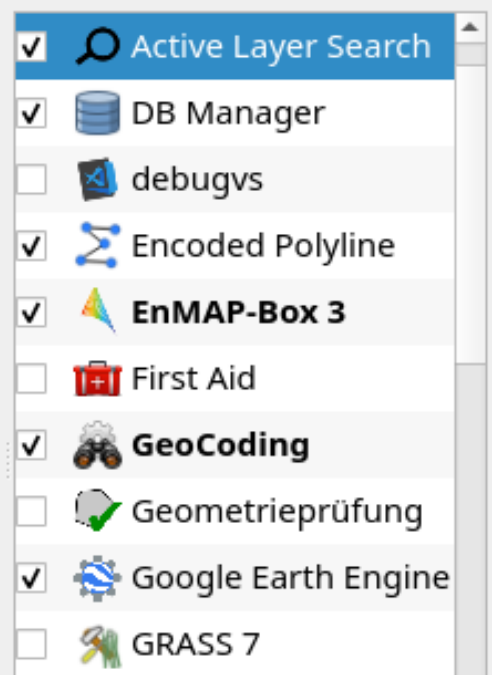
  → iface: ermöglicht es, auf die QGIS-Oberfläche zuzugreifen

- classFactory:

  → erhält von QGIS Zugriff auf das iface

  → Stellt das Plugin bereit

# __init__.py




Beim Start von QGIS:

- QGIS erstellt das *iface* und ermöglicht damit Plugins den Zugriff auf die QGIS-Oberfläche
- QGIS sucht für jedes aktivierte Plugin die *__init__.py*
- In der *__init__.py* sucht QGIS die Funktion *classFactory* und übergibt das *iface*

  → Beim Start gibt es immer mal wieder Python-Fehler (wenn ein aktiviertes Plugin fehlerhaft ist)

# active_layer_search.py

- Diese Datei enthält das eigentliche Plugin
- Der Name ist egal, aber es müssen 2 Methoden enthalten sein:
  - initGui

    → Wird ausgeführt, wenn das Plugin aktiviert wird (Häkchen wird gesetzt)

    → Platziert das Plugin in der QGIS-Oberfläche
  - unload

    → Wird ausgeführt, wenn das Plugin deaktiviert wird (Häkchen wird entfernt)

    → Entfernt das Plugin aus der QGIS-Oberfläche

# initGui (verpflichtend)

```python
# ActiveLayerSearch definiert das Plugin
class ActiveLayerSearch:

    # __init__ muss immer da sein
    # self.iface => Zugriff auf die Benutzeroberflaeche von QGIS
    def __init__(self, iface):
        self.iface = iface

    # Die Methode initGui muss immer da sein
    # Wird ausgefuehrt, wenn das Haekchen gesetzt wird
    def initGui(self):
        # Diese zwei Zeilen sorgen dafuer, dass das Plugin im Plugin Menu erscheint
        self.action_search = QAction('Suche starten', self.iface.mainWindow())
        self.iface.addPluginToMenu("&Active Layer Search", self.action_search)

        # Diese Zeile sorgt dafuer, dass das Plugin gestartet wird,
        # wenn man auf den Button klickt
        # Dabei wird die Methode start_search ausgefuehrt
        self.action_search.triggered.connect(self.start_search)
```

Legt fest, wie das Plugin in QGIS erscheint

Legt fest, dass das Plugin bei Klick auf einen Button gestartet wird

# initGui

Das schauen wir uns genauer an:

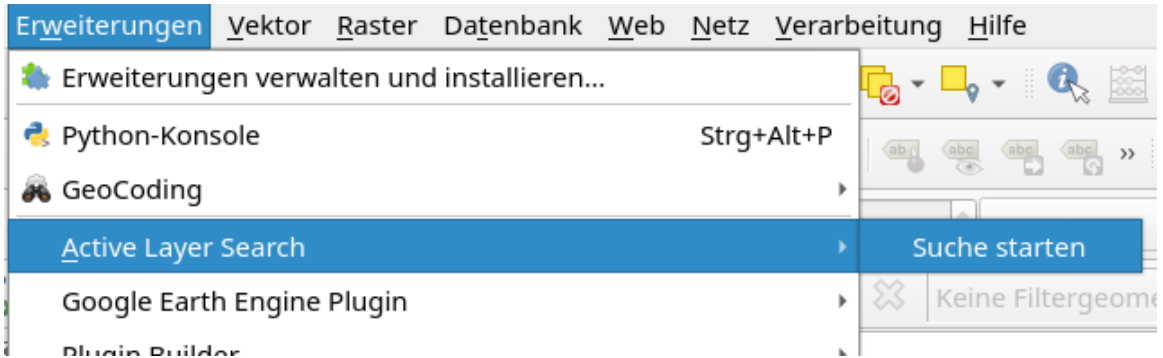
```
self.action_search = QAction('Suche starten', self.iface.mainWindow())

self.iface.addPluginToMenu("&Active Layer Search", self.action_search)
```

- QAction → PyQt Klasse, mit der ein Button ‚gebaut' werden kann
- self.action_search → Name des Buttons
- self.iface → Zugriff auf die QGIS Benutzeroberfläche
- addPluginToMenu → Befehl, mit dem der Button unter ‚Erweiterungen' im Menü erscheint

# initGui: Plugin in QGIS platzieren

Bemerkung:

- In dieser Version wird das Plugin nur unter ‚Erweiterungen' im Menü eingefügt
- Natürlich kann das Plugin auch an anderer Stelle erscheinen (zum Beispiel mit einem Icon in einer Toolbar)
- Wir werden die initGui später erweitern und  ein Icon verwenden



# initGui

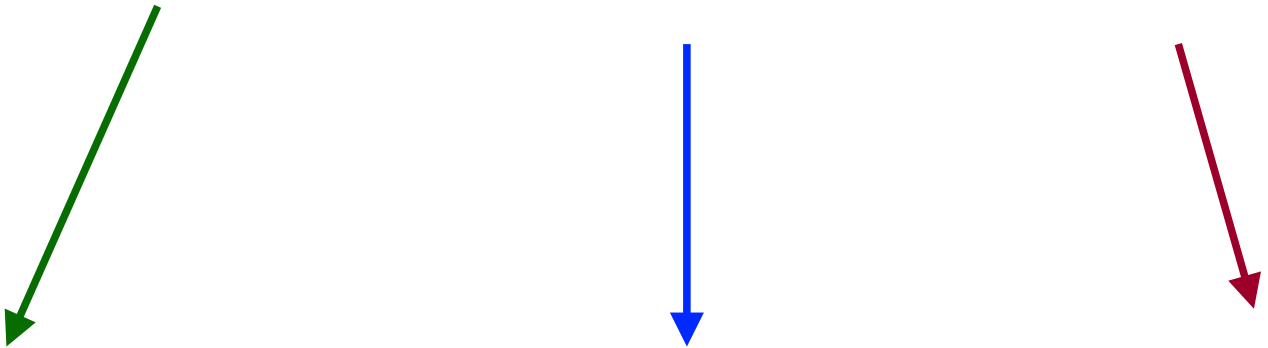
Und der Pluginstart:

```
self.action_search.triggered.connect(self.start_search)
```

- self.start_search → Methode, mit der das Plugin und die eigentliche Suchfunktion gestartet wird
- self.action_search → Button (siehe letzte Folie)
- self.action_search.triggerd → PyQt-Signal: immer wenn der Button getriggert wird (zum Beispiel beim einem Klick), wird die Methode self.start_search ausgeführt (das Plugin gestartet)

# PyQt-Signale verwenden: triggered

```
self.action_search.triggered.connect(self.start_search)
```



Name des Buttons

PyQt-Signal

Methode, die beim Klick ausgeführt wird

# start_search

```python
def start_search(self):
    """Wird ausgeführt beim Klick auf die Aktion (Menüeintrag)"""

    # Dialog erstellen
    search_dialog = SearchDialog()

    # Dialog zeigen / ausführen
    search_dialog.exec()
```

Die Gui wird gebaut

Die Gui wird angezeigt

# unload (verpflichtend)

```python
# Die Methode unload muss immer da sein
# Wird ausgefuehrt, wenn das Haekchen vom Plugin entfernt wird
def unload(self):
    self.iface.removePluginMenu("&Active Layer Search", self.action_search)
```

- Zweck: Entfernt das Plugin aus der QGIS-Oberfläche, wenn es deaktiviert wird
- Gegenteil von initGui
- self.iface → Zugriff auf die QGIS-Benutzeroberfläche

# search_dialog.ui

- Enthält die graphische Benutzeroberfläche
- XML-Struktur
- Kann mit dem Qt Designer erstellt und bearbeitet werden


search_dialog.ui

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>Dialog</class>
 <widget class="QDialog" name="Dialog">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>477</width>
    <height>200</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Layer durchsuchen</string>
  </property>
  <layout class="QVBoxLayout" name="verticalLayout"/>
 </widget>
 <resources/>
 <connections/>
</ui>
```

# Von der .ui zur fertigen GUI...

Hintergrund:

- Aus der .ui -Datei wird eine Klasse erzeugt
- Aus dieser Klasse wird die graphische Benutzeroberfläche(GUI) erzeugt
- **Dieser Code muss im Template nicht verändert werden, nur der Name der .ui-Datei wird angepass**t

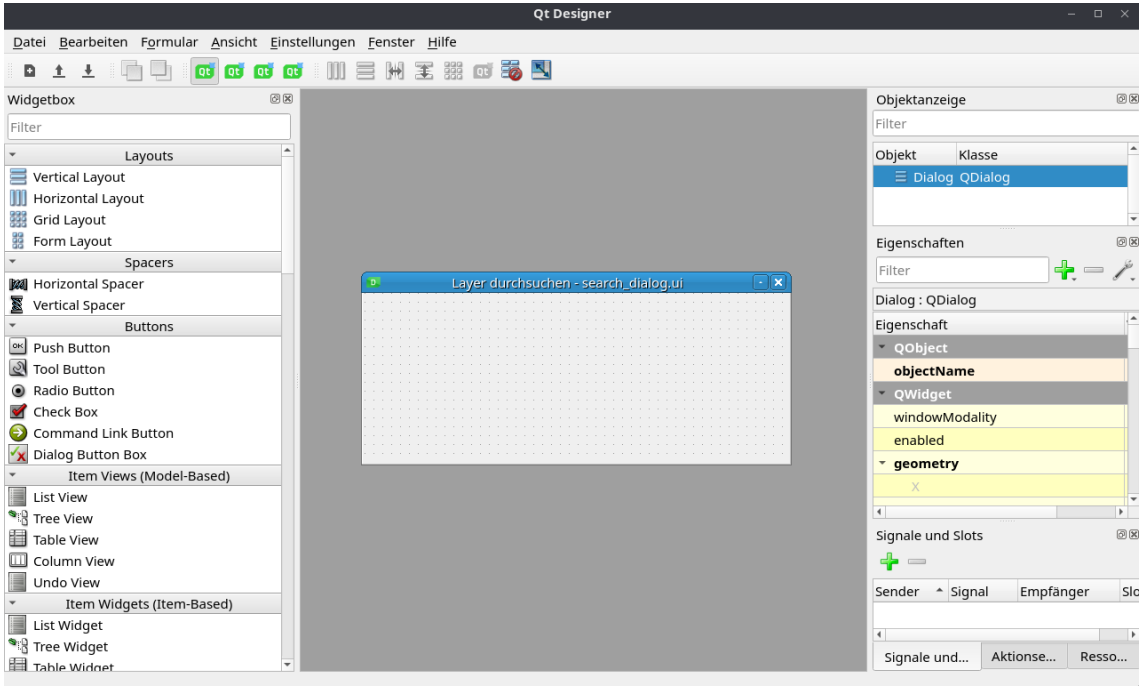
```
# Dialog aus .ui-Datei
WIDGET, BASE = uic.loadUiType(os.path.join(
    os.path.dirname(__file__), 'search_dialog.ui'))


class SearchDialog(BASE, WIDGET):

    def __init__(self, parent=None):
        super().__init__(parent)
        self.setupUi(self)
```

# Das war die Grundstruktur!

- Bei erneuter Verwendung des Templates sollten nur die Namen und Pfade angepasst werden
- Und… die eigentliche Funktionalität hinzufügen
- Wir haben vorhin ein Skript zur Suche im aktiven Layer geschrieben → In den Plugin Code integrieren!

# GUI erweitern: Qt Designer

Bitte öffnen Sie die Datei *search_dialog.ui* im Qt Designer:



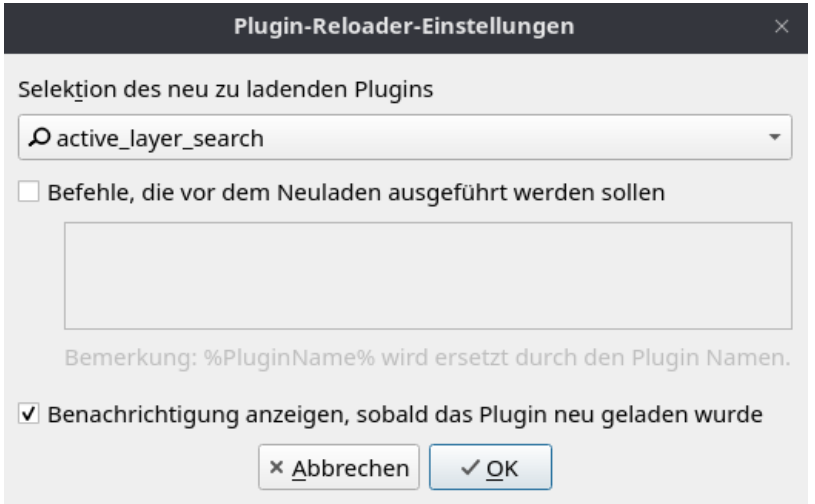
# GUI erweitern: Qt Designer


Bitte ergänzen Sie 3 Elemente:

- Display Widget: Label
- Input Widgets: Line Edit
- Buttons: Push Button

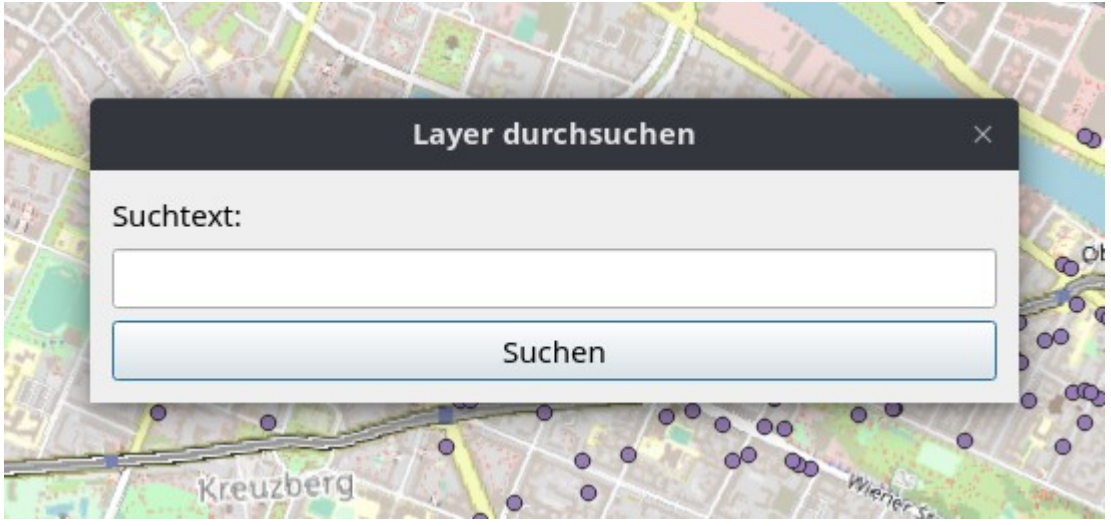
Bemerkung: Diese Elemente bekommen automatisch Objektnamen (rechte Seite: Eigenschaften → objectName)

# Plugin Reloader

# Graphische Benutzeroberfläche

→ Die Gui wurde erfolgreich angepasst!



# active_layer_search.py anpassen

```python
def start_search(self):
    """Wird ausgeführt beim Klick auf die Aktion (Menüeintrag)"""

    # Dialog erstellen
    self.search_dialog = SearchDialog()

    self.search_dialog.pushButton.clicked.connect(self.layer_search)

    # Dialog zeigen / ausführen
    self.search_dialog.exec()

def layer_search(self):

    print('Suche wird gestartet')
```

Suche durchführen:

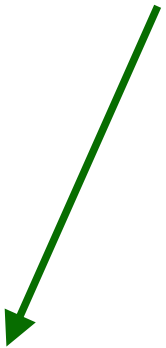
→ Bei einem Klick auf den Push Button soll die Suche durchgeführt werden

Bei einem Klick auf den Button wird die Methode *layer_search* ausgeführt

Die Methode *layer_search* wird definiert. Hier soll die eigentliche Suche stehen

# PyQt-Signale verwenden: clicked

```
self.search_dialog.pushButton.clicked.connect(self.layer_search)
```



Name des Push Buttons

PyQt-Signal

Methode, die beim Klick ausgeführt wird

# Erinnerung

```python
searchtext = 'Maison'

# iface -> Zugriff auf QGIS-Oberflaeche
# Aktiven Layer in lyr abspeichern
lyr = iface.activeLayer()

# Einzelne features aus dem Layer abspeichern
feats = lyr.getFeatures()

# Einzelne Features durchlaufen
for feat in feats:
    attrs = feat.attributes()

    # Suchtext verwenden
    if searchtext in str(attrs):

        # Zum gefundenen Feature zoomen
        # Feature ID berechnen
        fid = feat.id()
        # Feature selektieren
        lyr.selectByIds([fid])
        # Zum Feature zoomen
        iface.actionZoomToSelected().trigger()
        break
```

# active_layer_search.py anpassen

Die Nutzereingabe wird ausgelesen:

```
def layer_search(self):
    print('Suche wird gestartet')

    # Nutzereingabe auslesen
    searchtext = self.search_dialog.lineEdit.text()
    print(searchtext)
```

# active_layer_search.py anpassen

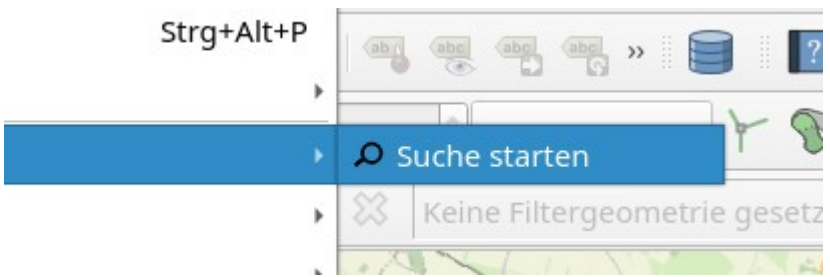
```python
def layer_search(self):

    # Nutzereingabe auslesen
    searchtext = self.search_dialog.lineEdit.text()

    # Skript einfuegen und anpassen
    lyr = self.iface.activeLayer()
    feats = lyr.getFeatures()

    for feat in feats:
        attrs = feat.attributes()
        if searchtext in str(attrs):
            fid = feat.id()
            lyr.selectByIds([fid])
            self.iface.actionZoomToSelected().trigger()
            break
```

iface → self.iface

# Icon einfügen

```
# Pfad zum Icon
icon_path = os.path.join(os.path.dirname(__file__), 'icon.svg')

# Diese zwei Zeilen sorgen dafuer, dass das Plugin im Plugin Menu erscheint
self.action_search = QAction(QIcon(icon_path), 'Suche starten', self.iface.mainWindow())
self.iface.addPluginToMenu("&Active Layer Search", self.action_search)
```

- Pfad zum Icon abspeichern
- QAction-Button anpassen



# Icon in der Toolbar einfügen

initGui:

```
self.iface.addToolBarIcon(self.action_search)
```

unload:

```
self.iface.removeToolBarIcon(self.action_search)
```

Fertig `:)`

# .zip Datei als Template

Die .zip Datei kann als Template verwendet werden. Was sollte man anpassen → Eigentlich nur die Namen :)

- Ordnername
- active_layer_search.py:
  - Namen der Datei und der Plugin-Klasse
  - Name der Gui-Klasse und der .ui Datei
- __init__.py
  - Pfade beim import und Namen des Plugins
- search_dialog.ui
  - Namen und Elemente

# .zip Datei als Template

Und dann muss man nur noch:

Neue Funktionen ergänzen :)

# Nützliche Links

PyQGIS Developer Cookbook

https://docs.qgis.org/3.28/en/docs/pyqgis_developer_cookbook/index.html

Ujaval Ghandi: Spatial Thoughts Blog

https://courses.spatialthoughts.com/pyqgis-in-a-day.html

Dokumentationen

PyQGIS: https://qgis.org/pyqgis/master/

PyQt: https://doc.qt.io/qtforpython-5/#documentation

# Vielen Dank für die Aufmerksamkeit!