

# **Geoprocessing mit Opensource Komponenten und Python**

Prof. Hans-Jörg Stark

Frobburgstrasse 24  
4052 Basel

[hjstark@proxxy.org](mailto:hjstark@proxxy.org)



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

## Abstract / Zusammenfassung

Der Kurs vermittelt die Grundlagen sowohl von Python in der Version 3.9.7 (oder 3.9.4) als auch der Geodatenverarbeitung mittels Python und entsprechenden Komponenten. Nach der erfolgreichen Bearbeitung dieser Kursunterlagen sind die Kursteilnehmenden in der Lage eigene Skripte zu erstellen, Python-Skripte lesen und je nach Komplexität selbständig anpassen zu können und die Grundprinzipien der Geodatenverarbeitung von Python verstanden zu haben. Sie kennen die wichtigsten Geodatenformate und können diese mit Python ansprechen und verarbeiten. Der Kurs vermittelt sowohl das Lesen als auch das Erzeugen bzw. Verändern von Geodaten (Vektor- und Rasterdaten) und deren einfache Visualisierung in einem Webbrowser.

# Inhaltsverzeichnis

	Seite
<b>1. Einführung .....</b>	<b>1</b>
1.1 Verwendete Quellen .....	1
<b>2. Geodaten im Tutorial .....</b>	<b>2</b>
2.1 Vektordaten .....	2
2.2 Rasterdaten.....	3
<b>3. Die Bibliothek GDAL/OGR .....</b>	<b>5</b>
3.1 Einsatz von OGR/GDAL über die Kommandozeile.....	7
3.2 Einsatz von OGR/GDAL mit Python .....	8
<b>4. Geodatenverarbeitung von Vektordaten.....</b>	<b>9</b>
4.1 Analyse Vektordaten.....	9
4.1.1 Analyse der Informationen eines gesamten Vektordatensatzes.....	9
4.1.2 Analyse der Informationen eines Objektes/Features eines Vektordatensatzes	11
4.2 Umprojektion .....	13
<b>5. Geodatenverarbeitung von Rasterdaten .....</b>	<b>14</b>
5.1 Extraktion von Basisinformationen einer Rasterdatei .....	14
5.2 Extraktion der radiometrischen Werte eines Pixels .....	18
5.3 Kommandozeilen-basierte Befehle zu Rasterdaten .....	20
5.4 Extraktion eines Teilbereichs aus einer Rasterdatei .....	21
<b>6. Geodaten schreiben.....</b>	<b>23</b>
6.1 Grundlegendes zum Schreiben von Geodaten mit GDAL/OGR.....	23
6.2 Setzen des Projektionssystems .....	23
6.3 Definition von Layername, Speicherort und Dateiformat.....	24
6.4 Definition der Attribute.....	25
6.5 Definition von Attributwerten und Geometrie eines Objekts.....	25
6.6 Umprojektion einer Ebene durch Umprojektion jedes einzelnen Datensatzes .....	29
6.7 Umprojektion von Rasterdaten.....	31
<b>7. Arbeiten mit PostgreSQL/PostGIS .....</b>	<b>33</b>
7.1 Laden von Beispieldaten nach PostgreSQL/PostGIS .....	33
7.2 Verbindung zur Datenbank .....	34
7.3 Abfrage - SQL Select Statement .....	35
7.4 Weitere Abfragen.....	35

7.5 Import von Daten mittels Python.....	36
<b>8. Aufruf und Verarbeiten von Geowebdiensten.....</b>	<b>38</b>
8.1 Grundlagen zu Geowebdiensten.....	38
8.2 Ansprechen von WMS und WFS mittels Python .....	39
8.2.1 WMS .....	39
8.2.2 WFS.....	41
<b>9. Arbeiten mit der Bibliothek Shapely.....</b>	<b>44</b>
<b>10. Arbeiten mit der Bibliothek Fiona .....</b>	<b>47</b>
<b>11. Visualisierung von Gedaten mit Folium .....</b>	<b>54</b>
<b>12. Abschlussübung .....</b>	<b>57</b>

# **Teil 1: Einführung in Python**

## **1. Einführung**

Die vorliegenden Kursunterlagen führen in die Möglichkeiten der Geodatenverarbeitung mit Open Source Komponenten mit Python ein. Sie sind keine vollständige und auch keine abschliessende Dokumentation und erheben daher keinen Anspruch auf Vollständigkeit. Der Fokus bei der Auswahl der Themen und der Tiefe der einzelnen Themenbereiche orientierte sich am zur Verfügung stehenden Zeitbudget des Kurses, der im Rahmen des ETH Moduls angeboten wird.

Die Kursteilnehmenden sollen möglichst viel selbst erarbeiten. Aus diesem Grund wurde die Theorie bewusst auf einem bescheidenen Niveau gehalten, um möglichst viel Zeit für Übungen und praktische Kursteile zu haben. Der Ansatz zur Vermittlung ist in dem Sinne "learning bei doing".

### **1.1 Verwendete Quellen**

Die hauptsächlich verwendeten Quellen sind folgende:

- Diener, Michael (2015). Python Geospatial Analysis Cookbook. Packt Publishing.
- Garrard, Chris (2016): Geoprocessing with Python. Manning Publications.
- Westra, Erik (2016): Python Geospatial Development. Third Edition. Packt Publishing.

## Teil 2: Einführung in die Geodatenverarbeitung mit Python

### 2. Geodaten im Tutorial

Die im Folgenden verwendeten Geodaten werden hier kurz vorgestellt. Es handelt sich dabei um Gemeinden des Kantons Solothurn, die kostenlos bezogen werden können<sup>1</sup>.

Im Verlauf des Kurses werden noch weitere Geodaten verarbeitet, die später eingeführt werden.

#### 2.1 Vektordaten

Die Gemeinden des Kantons Solothurn können direkt von der Webseite des Kantons bezogen werden. Sie sind im Folgenden in einer Visualisierung in QGIS dargestellt:

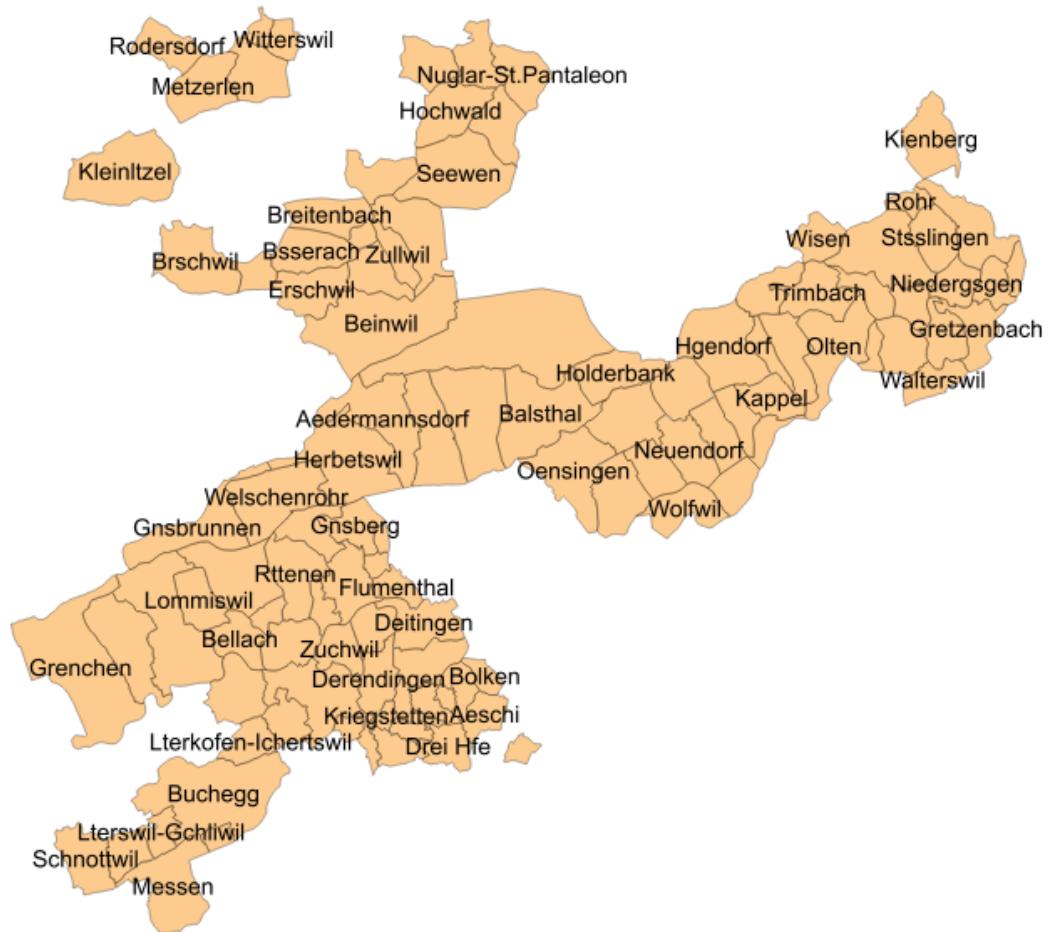


Abbildung 1 Gemeinden Kanton Solothurn

---

<sup>1</sup> <http://geoweb.so.ch/geodaten/index.php> (Stand 25. August 2016)

Die Gemeinden umfassen folgende Attribute am Beispiel der Gemeinde Rohr:

Attributabelle - public\_geo\_gemeinden :: Objekte gesamt: 109, gefiltert: 109, gewählt: 0

	<input type="checkbox"/> name
	Rohr
	Boningen
	Bsserach
	Egerkingen
	Walterswil
	Obergsgen
	Mmliswil-Ramiswil
	Subingen
	Oberdorf
	Rttenen
	Brschwil
	Beinwil
	Meltingen
	Zullwil
	Etziken
	Eppenberg-Wschnau
	Gnsberg
	Trimbach
	Matzendorf
	Kaopel
	<a href="#">Alle Objekte anzeigen</a>

## Abbildung 2 Attribute der Gemeinde Rohr (SO)

Mit den Gemeindedaten können bereits viele Anwendungsbeispiele simuliert und viele der Kernfunktionalitäten der Geodatenverarbeitung gezeigt werden.

## 2.2 Rasterdaten

Als Rasterdaten dient ein Orthofoto von Solothurn mit einer 12.5m Auflösung aus dem Jahr 2014, das ebenfalls von der Webseite des Kantons Solothurn kostenlos bezogen werden kann. Es ist in der folgenden Abbildung ebenfalls in QGIS dargestellt.



Abbildung 3 Luftbild von Solothurn (SO)

Werden die beiden Datensätze (Luftbild und Gemeinden) überlagert, präsentiert sich die Darstellung wie folgt:

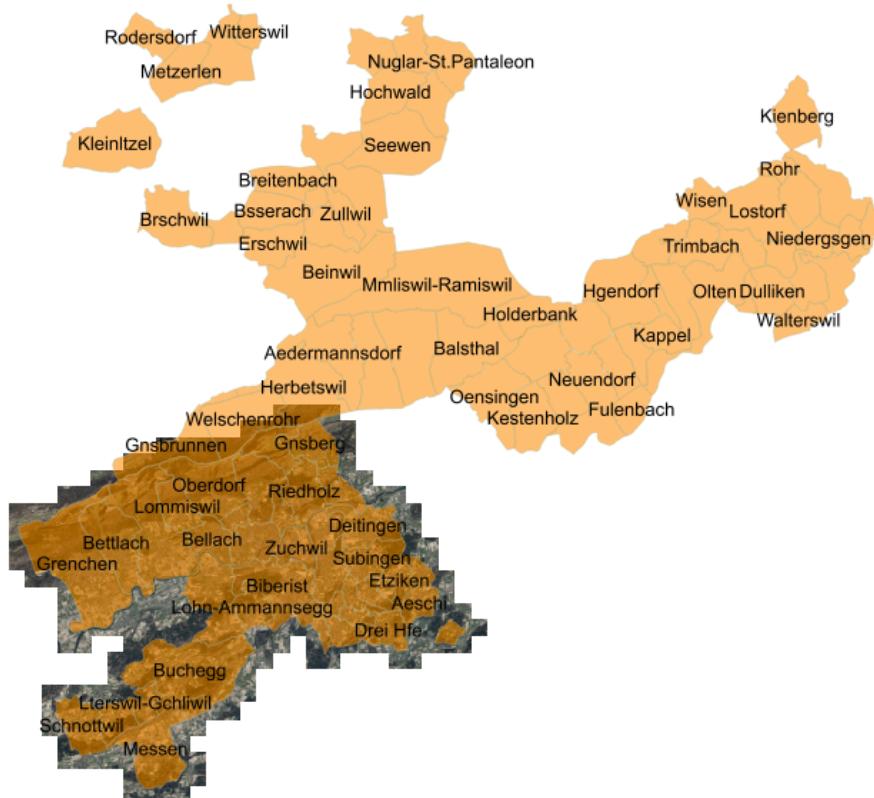


Abbildung 4 Kombinierte Darstellung der Daten von Solothurn

### 3. Die Bibliothek GDAL/OGR

Im Bereich von Opensource Komponenten werden einige Bibliotheken angeboten. Die wohl zZt umfangreichste und bekannteste ist GDAL/OGR. GDAL steht dabei für "Geospatial Data Abstraction Library". GDAL bietet Funktionalität für Rasterdaten, während OGR Funktionalität für Vektordaten anbietet. Ausführliche Informationen wie bspw. Erläuterung zu den Befehlen, Klassenbeschreibung usf. sind unter [www.gdal.org](http://www.gdal.org) zu finden.

GDAL/OGR bietet Lese- und Schreibzugriff auf über 200 Geodatenformate an. Die Bibliothek wird von diversen OpenSource Projekten genutzt, zu denen u.a. die folgenden zählen: GRASS, MapServer, Mapnik, QGIS, PostGIS, OTB, SAGA, FME, ArcGIS, Google Earth u.a.m.<sup>2</sup> Das Projekt startete 1998 und wurde von Frank Warmerdam entwickelt. Seit 2008 ist es ein offizielles OSGEO Projekt. GDAL/OGR steht unter einer MIT/X Open Source license zur Verfügung. Es handelt sich dabei um eine C++ library mit einer C API und wird von mehreren Betriebssystemen unterstützt. Es weist zahlreiche sog. Bindings zu Entwicklungsumgebungen auf, zu denen u.a. Python, Perl, C#, Java etc. gehören. OGC WKT coordinate systems und PROJ.4 werden als Kern für die Umprojektierung verwendet.<sup>3</sup>

GDAL bietet zahlreiche vordefinierte Befehle, die in der Konsole aufgerufen werden können. Diese sind unter [www.gdal.org/gdal\\_utilities.html](http://www.gdal.org/gdal_utilities.html) zu finden. Zu den bekanntesten zählen:

- **gdalinfo** - bietet Metadaten zu einer Rasterdatei
- **gdal\_translate** - kopiert einen Rasterdatensatz in ein anderes Format
- **gdalwarp** - projiziert einen Rasterdatensatz in ein anderes Bezugssystem um
- etc.

Für Vektordaten stehen ebenfalls gewisse vordefinierte Befehle zur Verfügung. Diese sind beschrieben unter [www.gdal.org/ogr\\_utilities.html](http://www.gdal.org/ogr_utilities.html). Zu den bekannten zählen:

- **ogrinfo** - bietet Metadaten zu einem Vektordatensatz
- **ogr2ogr** - Wandelt Simplefeatures-Daten von einem Format in ein anderes um

Als Formate werden wie bereits erwähnt viele und damit auch die gängisten unterstützt. Eine Auflistung ist zu finden unter:

---

<sup>2</sup> vgl. <http://trac.osgeo.org/gdal/wiki/SoftwareUsingGdal> (Stand 25. 8. 2016)

<sup>3</sup> vgl. [http://download.osgeo.org/gdal/presentations/GDAL%202.0%20overview%20\(FOSS4G-E%202015\).pdf](http://download.osgeo.org/gdal/presentations/GDAL%202.0%20overview%20(FOSS4G-E%202015).pdf) (Stand 25. 8. 2016)

Die aktuellste Version der Dokumentation von Änderungen kann unter

[http://download.osgeo.org/gdal/presentations/GDAL%202.1%20\(FOSS4G%20Bonn%202016\).pdf](http://download.osgeo.org/gdal/presentations/GDAL%202.1%20(FOSS4G%20Bonn%202016).pdf) (Stand 29. 8. 2016) aufgerufen werden

- [www.gdal.org/formats\\_list.html](http://www.gdal.org/formats_list.html) (Rasterdaten)
- [www.gdal.org/ogr\\_formats.html](http://www.gdal.org/ogr_formats.html) (Vektordaten)

Das Modell hinter GDAL/OGR sieht wie folgt aus:

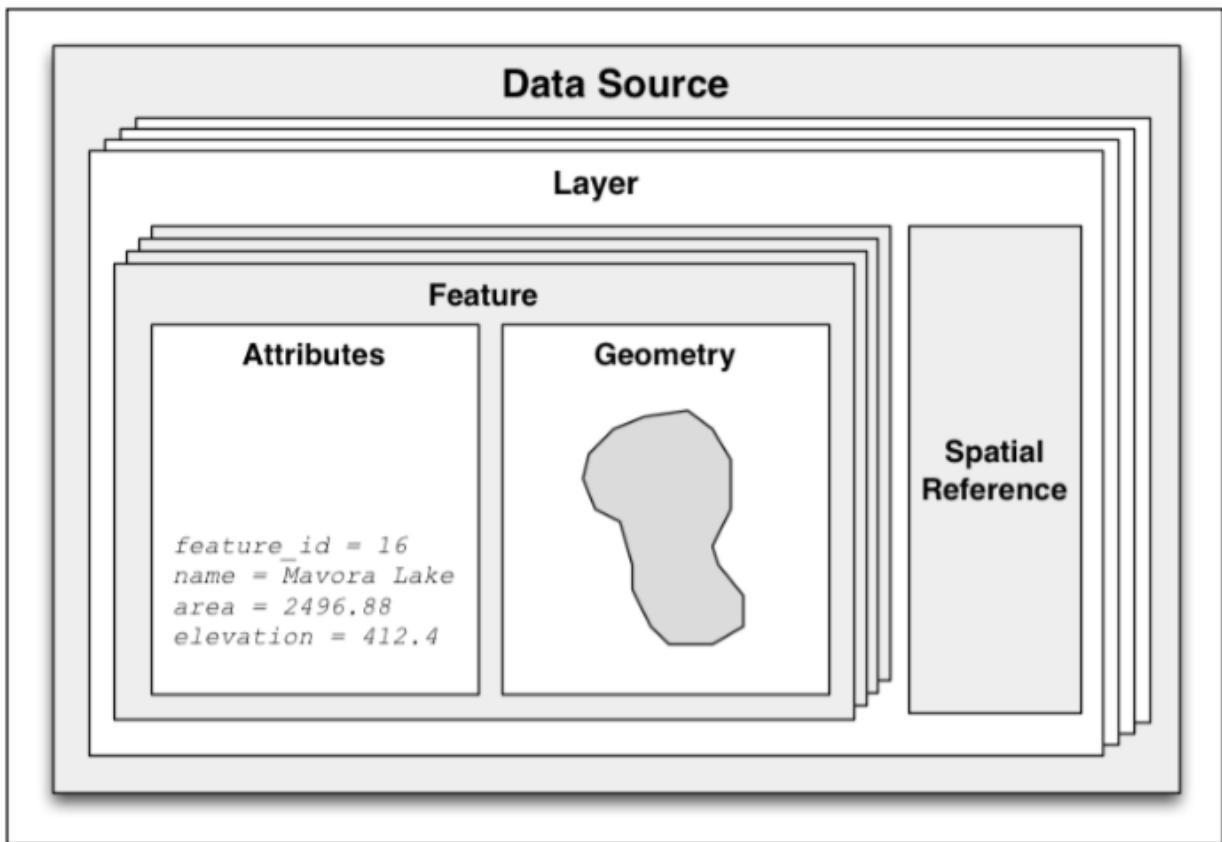


Abbildung 5 Architektur bzw. Komponenten von GDAL/OGR (Westra, S. 55)

Ein Datensatz hält also mindestens einen Layer, der seinerseits mindestens ein Feature (=Datensatz) enthält. Features sind durch Attribute beschrieben und enthalten eine Geometrie, die durch ein zugehöriges Räumliches Referenzsystem beschrieben sind.

### 3.1 Einsatz von OGR/GDAL über die Kommandozeile

GDAL und OGR können einfach für Standardaufgaben wie bspw. Formatumwandlungen ganzer Datensätze in der Konsole/Kommandozeile eingesetzt werden. Einige Beispielbefehle werden im Folgenden exemplarisch aufgelistet<sup>4</sup>:

#### Befehl

```
ogr2ogr --help
```

```
ogrinfo --help
```

```
ogr2ogr -f "ESRI Shapefile" mydata.shp  
mydata.tab
```

```
ogr2ogr -f "PostgreSQL" PG:"host=myhost  
user=myloginname dbname=mydbname  
password=mypassword" mytabfile.tab
```

```
ogr2ogr -f "PostgreSQL" PG:"host=myhost  
user=myloginname dbname=mydbname  
password=mypassword" mytabfile.tab -nln  
newtablename
```

```
ogr2ogr -f "ESRI Shapefile" mydata.shp  
PG:"host=myhost user=myloginname  
dbname=mydbname password=mypassword"  
"mytable"
```

```
ogr2ogr -f "KML" neighborhoods.kml  
PG:"host=myhost user=myloginname  
dbname=mydbname password=mypassword" -sql  
"select gid, name, the_geom from  
neighborhoods"
```

#### Erläuterung

Hilfe zu den erwähnten

Befehlen

Umwandlung eines MapInfo  
Datensatzes ins Format Esri  
Shape

Migration eines MapInfo  
Datensatzes in die  
Datenbank PostgreSQL

Migration eines MapInfo  
Datensatzes in die  
Datenbank PostgreSQL  
unter einem neuen Namen

Ausgabe eines Datensatzes  
von PostgreSQL ins Format  
Esri Shape

Ausgabe eines Datensatzes  
unter Einschränkung von  
bestimmten  
Feldern/Attributen von  
PostgreSQL ins Format KML

---

<sup>4</sup> vgl. [http://www.bostongis.com/?content\\_name=ogr\\_cheatsheet](http://www.bostongis.com/?content_name=ogr_cheatsheet) (Stand: 5. 9. 2016)

ogr2ogr -f "ESRI Shapefile" mydatadump	Export aller PostGIS
PG:"host=myhost user=myloginname	Tabellen in das Verzeichnis
dbname=mydbname password=mypassword"	mydatadump im Format
	Esri Shape

### 3.2 Einsatz von OGR/GDAL mit Python

Sollen die unter 3.1 erwähnten Kommandozeilenbefehle mittels Python verwendet werden, können diese unter Verwendung von `os.system()` wie folgt eingesetzt werden:

```
import os
os.system('ogr2ogr -f "ESRI Shapefile" -t_srs "EPSG:4326"
Daten/GEM_SO_WGS84.shp Daten/Gemeinden_Solothurn.shp -overwrite
-progress')
```

Hilfe zu einzelnen Funktionen, die GDAL/OGR anbieten, kann analog zur Hilfeinformation anderer Bibliotheken in Python abgefragt werden:

```
>>> from osgeo import ogr
>>> help(ogr.DataSource.CopyLayer)
```

## 4. Geodatenverarbeitung von Vektordaten

Zu Beginn der Geodatenverarbeitung werden Wege vorgestellt, um bestehende Vektordaten hinsichtlich ihrer

- Anzahl Datensätze
- Attribute (Namen und Werte)
- Räumliches Bezugssystem
- Räumliche Ausdehnung

zu analysieren.

### 4.1 Analyse Vektordaten

#### 4.1.1 Analyse der Informationen eines gesamten Vektordatensatzes

Damit mit der GDAL/OGR Bibliothek gearbeitet werden kann, muss sie zuerst in Python geladen werden. Dies erfolgt wie bei anderen Bibliotheken:

```
import ogr  
import os
```

Anschliessend sind die folgenden Schritte notwendig, um den Vektordatensatz zu analysieren:

- Bestimmen des Namens des Vektordatensatzes
- Bestimmen der Ablage des Vektordatensatzes
- Bestimmen des Dateiformates des Vektordatensatzes
- Öffnen des Vektordatensatzes

Diese Schritte sehen in Python wie folgt aus, wenn sie auf den Datensatz der Gemeinden von Solothurn angewendet werden, die im Esri Shape-Format vorliegen und in einem Unterverzeichnis Daten liegen:

```
drv = ogr.GetDriverByName("ESRI Shapefile")  
path2ds = os.path.join("Daten", "Gemeinden_Solothurn.shp")  
datasource = drv.Open(path2ds)
```

Wurden diese Teilaufgaben erfolgreich erledigt, können die weiter oben erwähnten Informationen abgefragt werden.

Damit die Anzahl der Gemeinden innerhalb des gewählten Datensatzes bestimmt werden können, muss gemäss dem OGR Architekturmodell zuerst noch der Layer definiert werden:

```
layer = datasource.GetLayer(0)
```

Anschliessend können die Anzahl Objekte bestimmt werden:

```
numftrs = layer.GetFeatureCount()
print ("Anzahl Gemeinden im Kanton Solothurn: %d" %numftrs)
```

Als nächstes werden die Anzahl Attribute und deren Namen ermittelt:

```
layer_def = layer.GetLayerDefn()
numatts = layer_def.GetFieldCount()
print("Anzahl Attribute: %d" %numatts)
```

Nachdem die Anzahl der Attribute bekannt ist, kann nun eine Schlaufe erstellt werden, mithilfe derer über alle Attribute iteriert wird und innerhalb welcher sowohl der Name des Attributs als auch dessen Typ ausgegeben werden:

```
for i in range(numatts):
    field_def = layer_def.GetFieldDefn(i)
    print(field_def.GetName())
    print(field_def.GetType())
    print(field_def.GetPrecision())
```

Damit diese Information etwas lesbarer ausgegeben wird, wird der oben stehende Code etwas angepasst:

```
for i in range(numatts):
    field_def = layer_def.GetFieldDefn(i)
    print("Attribut mit Bezeichnung <%s> vom Typ %s mit
Präzision %s" \
%(field_def.GetName(), field_def.GetType(), str(field_def.GetPrecision())))
```

Als nächstes wird das Räumliche Bezugssystem der Ebene bestimmt. Dieses gilt für alle Datensätze und wird mit dem Befehl GetSpatialRef() ermittelt.

```
lname = layer.GetName()
spatialRef = layer.GetSpatialRef()
print("Layer <%s> hat folgendes Räumliche Bezugssystem %s" %
(lname, spatialRef))
```

Als letztes wird die räumliche Ausdehnung des Datensatzes bestimmt. D.h. es werden die minimalen und die maximalen Koordinatenwerte sowohl in vertikaler als auch in horizontaler Richtung (dh. x- und y-Richtung) bestimmt. Dazu dient die Funktion GetExtent(). Diese Funktion liefert vier Werte zurück: minX, maxX, minY und maxY.

```
extent = layer.GetExtent()
```

```
print('Ausdehnung:', extent)
```

Als kleine **Übung** sollen die beiden Punkte ausgegeben werden, welche den minimalen und den maximalen Wert des Layer repräsentieren, dh. die Punkte unten links und oben rechts.

Nach dieser ersten Einführung soll nun folgende **Übung** selbstständig bearbeitet werden:  
Bestimme den Geometriertyp des Layers (Hinweis: Funktion GetGeomType ())

#### 4.1.2 Analyse der Informationen eines Objektes/Features eines Vektordatensatzes

So wie bisher Informationen zu einem gesamten Layer bestimmt wurden, können auch Informationen zu einem einzelnen Objekt eines Datensatzes (=Feature, Record) ermittelt werden.

Dazu zählen zB:

- Attributwerte
- Geometrie (Koordinaten)

Die Attributnamen sind ja aus dem Layer bereits bekannt. Nun können mithilfe dieser Namen die Attributwerte bestimmt werden. Damit dies erfolgreich gelingt, muss zuerst jedoch das einzelne Objekt/Feature ermittelt werden. Dies geschieht mit einem Index und der Funktion `GetFeature()`, die auf das Layerobjekt angewendet wird:

```
feature = layer.GetFeature(0)      #1. Feature
```

Die Ausgabe des Attributnamens einschließlich des zugehörigen Attributwerts für das spezifizierte Objekt kann folgendermassen erfolgen:

```
attributes = feature.items()
for key,value in attributes.items():
    print(" %s = %s" % (key, value))
```

---

<sup>5</sup> Quelle: <https://gist.github.com/walkermatt/7121427> (Stand: 26. 8. 2016)

Das Bestimmen der Geometrie einschliesslich der metrischen Informationen erfolgt über die Funktion `GetGeometryRef()`, welche direkt auf das feature-Objekt angewendet wird.

Als **Übung** soll die korrekte Anwendung auf das weiter oben bestimmte Feature (Gemeinde Rohr) wiederum selbst geschrieben werden.

Das Abrufen der Detailinformationen der Geometrie ist etwas komplexer. Hier muss beachtet werden, dass je nach Geometriertyp die Ausgabe umfangreicher ist. Dh. es muss ein Weg gesucht werden, welcher die Ausgabe der Geometrie dahingehend universell macht, dass der Quellcode unabhängig vom Geometriertyp verwendet werden kann. Eric Westra (S. 41) hat dazu ein gutes Code-Beispiel, das mit marginalen Änderungen hier wiedergegeben wird:

```
def analyzeGeometry(geometry, indent=0):
    s = []
    s.append(" " * indent)
    s.append(geometry.GetGeometryName())
    if geometry.GetPointCount() > 0:
        s.append(" mit %d Stuetzpunkten" %
geometry.GetPointCount())
    if geometry.GetGeometryCount() > 0:
        s.append(" enthaelt:")
    print "".join(s)

    for i in range(geometry.GetGeometryCount()):
        analyzeGeometry(geometry.GetGeometryRef(i), indent+1)
```

Diese Funktion kann einfach auf die Geometrie der Gemeinde Rohr angewendet werden:

```
analyzeGeometry(geometry)
```

Als nächste **Übung** soll aus den bisherigen einzelnen Schritten ein Skript erstellt werden..

Die nächste **Übung** besteht darin, den Datensatz auszutauschen: Statt die Gemeinden von Solothurn, sollen nun die Ländergrenzen als Basisdatensatz dienen. Dazu ist ein kostenloser Datensatz unter [http://thematicmapping.org/downloads/world\\_borders.php](http://thematicmapping.org/downloads/world_borders.php) verfügbar.

Als letzte **Übung** soll das Skript so angepasst werden, dass die Geometriedaten aller Gemeinden bzw. Länder ausgewertet werden.

Als optionale ***Übung*** soll das Skript so angepasst werden, dass der Datensatz, der ausgewertet werden soll, als Parameter dem Skript übergeben werden kann.

## 4.2 Umprojektion

Als weiteres Beispiel für den Umgang mit Vektordaten sollen die Koordinaten eines Punktes umprojiziert werden. Dazu dient der folgende Codeteil. Die Bibliothek `OSR`, die auch in GDAL/OGR enthalten ist, kommt hier zum Zug. Sie liefert die notwendige Funktionalität für das Umprojizieren der Koordinaten.

```
import ogr
import osr
source = osr.SpatialReference()
source.ImportFromEPSG(2927)

target = osr.SpatialReference()
target.ImportFromEPSG(4326)

transform = osr.CoordinateTransformation(source, target)

point = ogr.CreateGeometryFromWkt("POINT (1120351.57
741921.42)")
point.Transform(transform)
print(point.ExportToWkt())
```

## 5. Geodatenverarbeitung von Rasterdaten

Rasterdaten können nicht ganz so ausführlich wie Vektordaten ausgewertet werden. Dies liegt u.a. daran, dass sie keine Attributwerte beinhalten und hinsichtlich ihrer räumlichen Ausprägung einfacher als Vektordaten sind: Rasterdaten basieren auf Pixeln und sind daher per se stets flächenhafte Daten. Als einzige Information dient dem Pixel die radiometrische Auflösung, dh. der Farbwert des jeweiligen Pixels. Diese Information und andere können mit GDAL abgefragt werden. Zusammengefasst können folgende Informationen zu einer Rasterdatei extrahiert werden:

- Dimension
- Pixelgrösse
- Rotation
- Koordinatenwerte eines Referenzpixels
- Räumliches Bezugssystem
- Farbwerte von Pixeln der vorhandenen Kanälen

### 5.1 Extraktion von Basisinformationen einer Rasterdatei

Für die Extraktion von Basisinformationen einer Rasterdatei muss zusätzlich die Klasse `gdalconst` importiert werden. Die Anwendung in Python von GDAL funktioniert bei Rasterdaten grundsätzlich analog zur Anwendung bei Vektordaten mit OGR. Allerdings müssen zuerst die notwendigen Treiber registriert werden, bevor sie benutzt werden können. Erste einfache Informationen sind auf Kommandozeilen-Ebene mit dem Befehl `gdalinfo <Rasterdateiname>` erfahrbar:

```
gdalinfo ortho14_5m_rgb_solothurn.tif
```

Dieser Befehl führt zu folgender Ausgabe:

```
[Hans-Jorgs-MacBook-Pro:Jupyter hansjoerg.stark$ gdalinfo ../Daten/ortho14_5m_rgb_solothurn.tif
Driver: GTiff/GeoTIFF
Files: ../Daten/ortho14_5m_rgb_solothurn.tif
Size is 5800, 4800
Coordinate System is:
PROJCS["CH1903 / LV03",
    GEOGCS["CH1903",
        DATUM["CH1903",
            SPHEROID("Bessel 1841",6377397.155,299.1528128000008,
                AUTHORITY["EPSG","7004"]),
            TOWGS84[674.4,15.1,405.3,0,0,0,0],
            AUTHORITY["EPSG","6149"]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433],
        AUTHORITY["EPSG","4149"]],
    PROJECTION["Hotine_Obllique_Mercator_Azimuth_Center"],
    PARAMETER["latitude_of_center",46.95240555555556],
    PARAMETER["longitude_of_center",7.439583333333333],
    PARAMETER["azimuth",90],
    PARAMETER["rectified_grid_angle",90],
    PARAMETER["scale_factor",1],
    PARAMETER["false_easting",600000],
    PARAMETER["false_northing",200000],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AUTHORITY["EPSG","21781"]]
Origin = (592000.0000000000000000,237000.0000000000000000)
Pixel Size = (5.000000000000000,-5.000000000000000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  COMPRESSION=LZW
  INTERLEAVE=PIXEL
Corner Coordinates:
Upper Left  ( 592000.000, 237000.000) ( 7d20' 1.76"E, 47d17' 6.74"N)
Lower Left  ( 592000.000, 213000.000) ( 7d20' 3.30"E, 47d 4' 9.51"N)
Upper Right ( 621000.000, 237000.000) ( 7d43' 1.94"E, 47d17' 5.71"N)
Lower Right ( 621000.000, 213000.000) ( 7d42'57.91"E, 47d 4' 8.48"N)
Center      ( 606500.000, 225000.000) ( 7d31'31.23"E, 47d10'38.19"N)
Band 1 Block=256x256 Type=Byte, ColorInterp=Red
  Overviews: 2900x2400, 1450x1200, 725x600, 363x300, 182x150, 91x75, 46x38
Band 2 Block=256x256 Type=Byte, ColorInterp=Green
  Overviews: 2900x2400, 1450x1200, 725x600, 363x300, 182x150, 91x75, 46x38
Band 3 Block=256x256 Type=Byte, ColorInterp=Blue
  Overviews: 2900x2400, 1450x1200, 725x600, 363x300, 182x150, 91x75, 46x38
```

Abbildung 6 Ausgabe von gdalinfo()

Einige dieser Infos sollen nun über Python extrahiert werden. Zuerst werden die Anzahl Spalten, Zeilen und Bänder des Rasterbildes abgefragt:

```
import gdal

fn = 'Daten/ortho14_5m_rgb_solothurn.tif'
ds = gdal.Open(fn)
if ds is None:
    print ('Datensatz %s konnte nicht geöffnet werden!' %fn)
    sys.exit(1)

#Dimension des Rasterbildes
cols = ds.RasterXSize
rows = ds.RasterYSize
bands = ds.RasterCount

print ("Anzahl Spalten: %d" %cols)
print ("Anzahl Zeilen: %d" %rows)
print ("Anzahl Baender: %d" %bands)
```

Gemäss [http://www.gdal.org/gdal\\_tutorial.html](http://www.gdal.org/gdal_tutorial.html) sind folgende Meta-Infos zum Datensatz extrahierbar, welche die Lageinformationen zum Rasterbild ausgeben:

```
adfGeoTransform[0] /* top left x */
adfGeoTransform[1] /* w-e pixel resolution */
adfGeoTransform[2] /* rotation, 0 if image is "north up" */
adfGeoTransform[3] /* top left y */
adfGeoTransform[4] /* rotation, 0 if image is "north up" */
adfGeoTransform[5] /* n-s pixel resolution */
```

Diese Informationen werden nun mittels Python ausgeschöpft:

```
geotransform = ds.GetGeoTransform()
if not geotransform is None:
    print ('Origin = (',geotransform[0],
',',geotransform[3],')')
    print ('Pixel Size = (',geotransform[1],
',',geotransform[5],')')
    print ('Rotation: ',geotransform[2], ' /
',geotransform[4],')')
```

Die bisherigen Informationen bezogen sich auf die gesamte Rasterdatei, unabhängig vom jeweiligen Farbband. Um nun nähere Angaben zu den Farbbändern zu erhalten, dient der folgende Quellcode:

```
#Bandinformationen
band = ds.GetRasterBand(1)
print ('Band-Typ: ',gdal.GetDataTypeName(band.DataType))

min = band.GetMinimum()
max = band.GetMaximum()
if min is None or max is None:
    (min,max) = band.ComputeRasterMinMax(1)
print ('Min=% .3f, Max=% .3f' % (min,max))

if band.GetOverviewCount() > 0:
    print ('Band hat ', band.GetOverviewCount(), '
Übersichten.')

if not band.GetRasterColorTable() is None:
    print ('Band hat eine Farbtabelle mit ', \
band.GetRasterColorTable().GetCount(), ' Einträgen.') 
```

Diese einzelnen Code-Bestandteile werden nun zu einem gesamten Skript zusammengefasst:

```
import gdal

fn = 'Daten/ortho14_5m_rgb_solothurn.tif'
ds = gdal.Open(fn)
if ds is None:
    print ('Datensatz %s konnte nicht geöffnet werden!' %fn)
    sys.exit(1)

#Dimension des Rasterbildes
cols = ds.RasterXSize
rows = ds.RasterYSize
bands = ds.RasterCount

print ("Anzahl Spalten: %d" %cols)
print ("Anzahl Zeilen: %d" %rows)
print ("Anzahl Baender: %d" %bands)

geotransform = ds.GetGeoTransform()
if not geotransform is None:
    print ('Origin = (' ,geotransform[0],
',',geotransform[3],',')
    print ('Pixel Size = (' ,geotransform[1],
',',geotransform[5],',')
    print ('Rotation: ',geotransform[2], ' /
',geotransform[4],')')

#Bandinformationen
band = ds.GetRasterBand(1)
print ('Band-Typ: ',gdal.GetDataTypeName(band.DataType))

min = band.GetMinimum()
max = band.GetMaximum()
if min is None or max is None:
    (min,max) = band.ComputeRasterMinMax(1)
print ('Min=% .3f, Max=% .3f' % (min,max))

if band.GetOverviewCount() > 0:
    print ('Band hat ', band.GetOverviewCount(), ' Übersichten.')

if not band.GetRasterColorTable() is None:
    print ('Band hat eine Farbtabelle mit ', \
band.GetRasterColorTable().GetCount(), ' Einträgen.')
```

Als **Übung** soll nun die Rasterinformation nicht nur für ein Band, sondern für alle im Bild vorhandenen Bänder ausgegeben werden.

## 5.2 Extraktion der radiometrischen Werte eines Pixels

Es kann von besonderem Interesse sein, die "Farbwerte" an bestimmten Stellen eines Gebiets, für welches Rasterdaten vorliegen, abzufragen. Dh. es müssen die Pixelwerte der einzelnen Bänder eines bestimmten Pixels in einer Rasterdatei abgefragt werden. So könnten zB in einer Rasterdatei, welche den Höhenwert als Pixelwert enthält, einzelne Punkte auf ihren Höhenwert hin untersucht werden. Um dies zu tun, müssen die Koordinatenwerte der Untersuchungspunkte zuerst in den Bildraum transformiert werden. Dort werden dann die entsprechenden Zeilen- und Spaltenindices der Rasterdatei bestimmt, um anhand dieser die Pixelwerte abzufragen.

Mittels folgendem Code werden die Farbwerte abgefragt, nachdem die Pixelkoordinaten bekannt sind:

```
data = band.ReadAsArray(xPixelOffset, yPixelOffset, 1, 1)
value = data[0,0]
```

Dabei entspricht die Syntax des ReadAsArray-Befehls der folgenden Bedeutung:

ReadAsArray(<xoff>, <yoff>, <xsize>, <ysize>)

Im Folgenden wird ein Wert an drei Stellen abgefragt:

```
# obtained from
http://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides
4.pdf and adapted
# script to get pixel values at a set of coordinate by reading
in one pixel at a time

import os, sys, numpy, time
from osgeo import gdal
from osgeo.gdalconst import *

# start timing
startTime = time.time()
# coordinates to get pixel values for
xValues = [594000.0, 604000.0, 613500.0]
yValues = [229500.0, 231000.0, 222800.0]

# register all of the drivers
gdal.AllRegister()
# open the image
ds = gdal.Open('Daten/ortho14_5m_rgb_solothurn.tif',
GA_ReadOnly)
if ds is None:
    print ('Could not open image')
    sys.exit()
```

```

# get image size
rows = ds.RasterYSize
cols = ds.RasterXSize
bands = ds.RasterCount
# get georeference info
transform = ds.GetGeoTransform()
xOrigin = transform[0]
yOrigin = transform[3]
pixelWidth = transform[1]
pixelHeight = transform[5]
print('X | Y | xOffset | yOffset | Wert Band 1 | Wert Band 2 |'
      'Wert Band 3 ')

# loop through the coordinates
for i in range(3):
    # get x,y
    x = xValues[i]
    y = yValues[i]
    # compute pixel offset
    xOffset = int((x - xOrigin) / pixelWidth)
    yOffset = int((y - yOrigin) / pixelHeight)
    # create a string to print out
    s = str(x) + ' ' + str(y) + ' ' + str(xOffset) + ' ' +
str(yOffset) + ' '
    # loop through the bands
    for j in range(bands):
        band = ds.GetRasterBand(j+1) # 1-based index
        #read data and add the value to the string
        data = band.ReadAsArray(xOffset, yOffset, 1, 1)
        value = data[0,0]
        s = s + str(value) + ' '

        if not band.GetRasterColorTable() is None:
            print ('Band has a color table with ', \
                  band.GetRasterColorTable().GetCount(), ' entries.')
    #print out the data string
    print(s)
# figure out how long the script took to run
endTime = time.time()
print()
print ('The script took ' + str(endTime - startTime) + ' '
      'seconds')

```

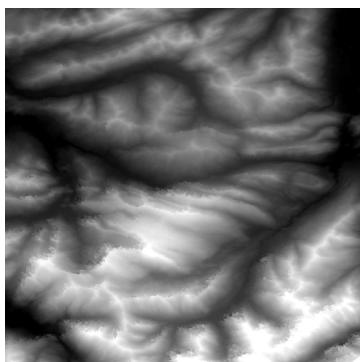
### 5.3 Kommandozeilen-basierte Befehle zu Rasterdaten

GDLA bietet auch diverse, vordefinierte Kommandozeilen-Befehle. Einige wenige werden hier vorgestellt:

Als Beispiel sollen aus einem digitalen Höhenmodell (DEM), das als GeoTIFF vorliegt, Kontur-Linien berechnet werden. Der Befehl dazu lautet:

```
gdal_contour -a elevation -i 100.0 Daten/Elevation_raster.tif  
contour_100.shp
```

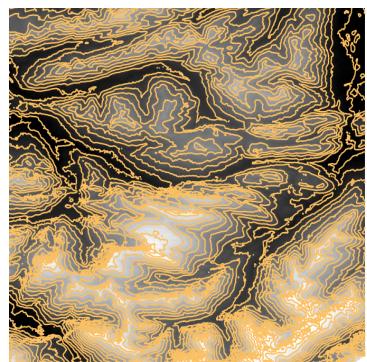
Die folgenden drei Abbildungen zeigen abgeleitete Produkte aus einem Höhenmodell (als Rasterdatei):<sup>6</sup>



Ausgangs-DEM



Konturlinien

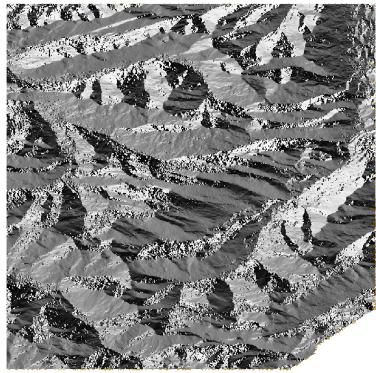


DEM überlagert mit Konturlinien

---

<sup>6</sup> Diese Ansichten können gut in QGIS nachvollzogen werden

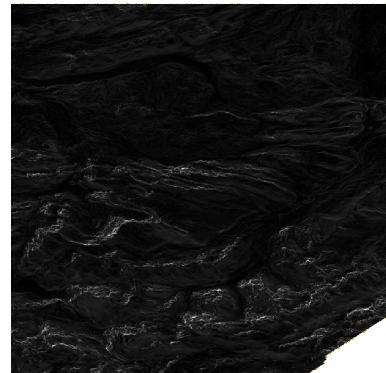
Mit dem Befehl gdaldem können weitere Produkte rund um Digitale Höhenmodelle erzeugt werden (bspw. Exposition- und Neigungskarten):



Ausrichtung



Schattierung



Neigung

Die entsprechenden Befehle dazu lauten:

Ausrichtung: gdaldem aspect

Schattierung: gdaldem hillshade

Neigung: gdaldem slope

Als **Übung** soll folgendes erstellt werden: Erzeuge eine Punktliste mit zufällig verteilten Punkten, die alle innerhalb des Höhenmodells "Elevation\_Raster.tif" liegen und bestimme für jeden dieser Punkte den Höhenwert, die Ausrichtung und die Neigung.

## 5.4 Extraktion eines Teilbereichs aus einer Rasterdatei

Es kann sein, dass das Bedürfnis besteht, aus einer bestehenden Rasterdatei einen Bereich auszuschneiden und in einer separaten Rasterdatei abzuspeichern. Dazu kann einer der erwähnten Kommandozeilenbefehle - gdal\_translate mit der Option -projwin - verwendet werden.

Das folgende Beispiel schneidet aus der Datei world.png den ungefähren Bereich von Europa aus und speichert diesen in einer separaten Datei und im Format TIFF; es werden zwei Dateien für den Bereich Europa erstellt: eine 1:1 Kopie des Ausschnitts und eine, die um  $\frac{1}{4}$  kleiner ist (jeweils 50% in beide Achsenrichtungen):

```
import os, sys, numpy, time
from osgeo import gdal
from osgeo.gdalconst import *
import csv
```

```

# register all of the drivers
gdal.AllRegister()

#Open Rasterfile
fn = 'Daten/worldmap.jpg'
ds = gdal.Open(fn)
if ds is None:
    print ('Datensatz %s konnte nicht geöffnet werden' %fn)
    sys.exit(1)

os.system('gdalinfo %s' %fn)
translatecommand = 'gdal_translate -projwin 1680 170 2200 550 \
%s Daten/europe.tif' %fn
print ("command to run: %s" %translatecommand )
os.system(translatecommand)

#kleinere Kopie von Europa
translatecommand = 'gdal_translate -projwin 1680 170 2200 550 - \
outsize 50% 50% %s Daten/europesmall.tif' %fn
print ("command to run: %s" %translatecommand)
os.system(translatecommand)

print ("")
os.system('gdalinfo Daten/europe.tif')
print ("")
os.system('gdalinfo Daten/europesmall.tif')

```

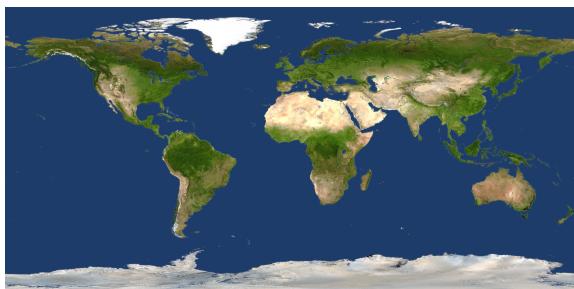


Abbildung 7 Worldmap<sup>7</sup>



Abbildung 8 Europa

---

<sup>7</sup> <http://www.jaas.de/img/wom/worldmap2.jpg> (29. 8. 2016)

## 6. Geodaten schreiben

### 6.1 Grundlegendes zum Schreiben von Geodaten mit GDAL/OGR

Bisher wurden Geodaten ausschliesslich analysiert. Es wurden Informationen aus vorliegenden Geodaten extrahiert und ausgegeben. Dazu gehörten Attributnamen und -werte, Informationen zum Projektionssystem oder die Ausdehnung usf. In diesem Kapitel sollen nun Geodaten geschrieben, dh. erstellt werden. Analog zur Analyse existierender Daten können wichtige Schritte bei der Erstellung einer neuen Ebene konzeptionell festgehalten werden. Zu diesen zählen in Bezug auf den zu erstellenden Datensatz die Definition bzw. Festlegung von:

- Namen
- Speicherort
- Format
- Räumliches Bezugssystem
- Räumlicher Datentyp
- Attributnamen
- Attributdatentypen und ggf -länge
- Ev. Wertebereiche

In Bezug auf die einzelnen Objekte / Features müssen schliesslich die folgenden Informationen bekannt sein und verarbeitet werden:

- Attributwerte
- Primäre Metrik (Koordinaten)
- Falls nötig: Fremdschlüssel für Verlinkung mit anderen Datensätzen

### 6.2 Setzen des Projektionssystems

Um das Projektionssystem der zu erstellenden Ebene festzulegen, dient das Modul OSR. Dieses bietet die notwendigen Methoden für die Festlegung und das Abrufen der Informationen zum räumlichen Bezugssystem:

```
import osr

srs = osr.SpatialReference()
srs.SetWellKnownGeogCS('WGS84')
#srs.SetFromUserInput("EPSG:21781")

print(srs.GetAttrValue("AUTHORITY", 0))
print(srs.GetAttrValue("AUTHORITY", 1))

print()
```

```

prj1=srs.ExportToWkt()
print(prj1)

print()
prj2=srs.ExportToPrettyWkt()
print(prj2)

print()
prj3=srs.ExportToProj4()
print(prj3)

```

Alternativ kann auch das Projektionssystem aus einem bestehenden Layer übernommen werden:

```

shapefile = osgeo.ogr.Open("GEMEINDEN_BL.shp")
layer = shapefile.GetLayer(0)
srs = osr.SpatialReference()
srs.ImportFromProj4(layer.GetSpatialRef().ExportToProj4())

```

### 6.3 Definition von Layername, Speicherort und Dateiformat

Als erstes werden der Dateiname, der Speicherort und das Dateiformat des zu erstellenden Datensatzes definiert. Dazu dienen folgende Befehle:

```

import ogr,os

driver = ogr.GetDriverByName("Esri Shapefile")
#driver = ogr.GetDriverByName("GeoJSON")

if os.path.exists("Daten/NeuerLayer.shp"):
    driver.DeleteDataSource("Daten/NeuerLayer.shp")

destinationFile =
driver.CreateDataSource("Daten/NeuerLayer.shp")
#destinationFile =
driver.CreateDataSource("Daten/NeuerLayer.geojson")

destinationLayer = destinationFile.CreateLayer("Layer", srs)

```

## 6.4 Definition der Attribute

Nach der Erstellung des neuen Datensatzes müssen die Attribute im festgelegt werden. Im Folgenden werden zwei neue Spalten definiert: Name vom Typ String mit der maximalen Länge von 80 Zeichen und Bemerkung vom Typ String mit der maximalen Länge von 80 Zeichen. Diesen können bspw. wie folgt definiert werden:

```
fieldDef = ogr.FieldDefn("Name",ogr.OFTString)
fieldDef.setWidth(80)
destinationLayer.CreateField(fieldDef)
fieldDef = ogr.FieldDefn("Bemerkung", ogr.OFTString)
fieldDef.setWidth(100)
destinationLayer.CreateField(fieldDef)
```

Manchmal kann es sein, dass die gesamte Dateistruktur eines bestehenden Datensatzes übernommen werden soll. In diesem Fall wird zuerst der Ursprungsdatensatz (sourcelayer) geöffnet und anschliessend über jedes der vorhandenen Attribute iteriert und deren Definition dem Definitionsobjekt für den neu zu erstellenden Datensatz hinzugefügt:

Falls die Attribut-Struktur einer bestehenden Ebene gleich übernommen werden soll für den Ziel-Layer kann dies wie folgt umgesetzt werden:

```
for i in range(sourcelayer.GetLayerDefn().GetFieldCount()):
    destinationLayer.CreateField(sourcelayer.GetLayerDefn().GetFieldDefn(i))
```

## 6.5 Definition von Attributwerten und Geometrie eines Objekts

Nachdem der Datensatz dahingehend vorbereitet ist, dass die Attribute definiert wurden, können nun Objekte in die Ebene eingefügt werden. Dieser Schritt ist etwas aufwändiger als die bisherigen, da die genaue Geometrie sauber spezifiziert werden muss. Anhand eines Polygons wird dieser Schritt im Folgenden illustriert:

```

ftrName = "Erstes Features"

#geometry = feature.GetGeometryRef()
minEasting = 10
maxEasting = 20
minNorthing = 15
maxNorthing = 25

#Definition des OGR Geometrieobjekts als LinearRing
linearRing = ogr.Geometry(ogr.wkbLinearRing)
#Hinzufügen der Stützpunkte des LinearRing
linearRing.AddPoint(minEasting, minNorthing)
linearRing.AddPoint(maxEasting, minNorthing)
linearRing.AddPoint(maxEasting, maxNorthing)
linearRing.AddPoint(minEasting, maxNorthing)
linearRing.AddPoint(minEasting, minNorthing)

#Instanzieren der Geometrie als WKBPolygon ins sqr Objekt
sqr = ogr.Geometry(ogr.wkbPolygon)

#Zuweisen der Geometrie zum instanzierten Objekt
sqr.AddGeometry(linearRing)

#Neues Feature erhält Attributdefinition
sqrfeature = ogr.Feature(destinationLayer.GetLayerDefn())

#Neues Feature erhält Geometrie
sqrfeature.SetGeometry(sqr)

#Neues Feature erhält für das Attribut Name den Wert "Erstes
Feature"
sqrfeature.SetField("Name", ftrName)

#Erstellung des Features im neuen Layer
destinationLayer.CreateFeature(sqrfeature)

#Freigabe des Featureobjekts
sqrfeature.Destroy()

```

Das gesamte Skript präsentiert sich am Ende wie folgt:

```
import ogr,os,osr

#Definition SRS
srs = osr.SpatialReference()
srs.SetWellKnownGeogCS('WGS84')

#Erstellen der neuen Ebene/Layer
#driver = ogr.GetDriverByName("Esri Shapefile")
driver = ogr.GetDriverByName("GeoJSON")
if os.path.exists("Daten/NeuerLayer.shp"):
    driver.DeleteDataSource("Daten/NeuerLayer.shp")
#destinationFile =
driver.CreateDataSource("Daten/NeuerLayer.shp")
destinationFile =
driver.CreateDataSource("Daten/NeuerLayer.geojson")
destinationLayer = destinationFile.CreateLayer("Layer", srs)

#Festlegung der Attribute
fieldDef = ogr.FieldDefn('Id', ogr.OFTInteger)
destinationLayer.CreateField(fieldDef)
fieldDef = ogr.FieldDefn("Name",ogr.OFTString)
fieldDef.setWidth(80)
destinationLayer.CreateField(fieldDef)
fieldDef = ogr.FieldDefn("Bemerkung", ogr.OFTString)
fieldDef.setWidth(100)
destinationLayer.CreateField(fieldDef)

#Erstellen eines Features
ftrName = "Erstes Feature"
ftrBem = "Dies ist mein erstes selbst erstelltes Features"

#geometry = feature.GetGeometryRef()
minEasting = 10
maxEasting = 20
minNorthing = 15
maxNorthing = 25

#Definition des OGR Geometrieobjekts als LinearRing
linearRing = ogr.Geometry(ogr.wkbLinearRing)
#Hinzufügen der Stützpunkte des LinearRing
linearRing.AddPoint(minEasting, minNorthing)
linearRing.AddPoint(maxEasting, minNorthing)
linearRing.AddPoint(maxEasting, maxNorthing)
linearRing.AddPoint(minEasting, maxNorthing)
linearRing.AddPoint(minEasting, minNorthing)

#Instanzieren der Geometrie als WKBPolygon ins sqr Objekt
sqr = ogr.Geometry(ogr.wkbPolygon)
#Zuweisen der Geometrie zum instanzierten Objekt
```

```

sqr.AddGeometry(linearRing)
#Neues Feature erhält Attributdefinition
sqrfeature = ogr.Feature(destinationLayer.GetLayerDefn())
#Neues Feature erhält Geometrie
sqrfeature.SetGeometry(sqr)
#Neues Feature erhält für das Attribut Name den Wert "Erstes
Feature"
sqrfeature.SetField("Id", 1)
sqrfeature.SetField("Name", ftrName)
sqrfeature.SetField("Bemerkung", ftrBem)
#Erstellung des Features im neuen Layer
destinationLayer.CreateFeature(sqrfeature)
sqrfeature.Destroy()

print("Erstellung abgeschlossen")
destinationFile.Destroy()

```

Im Zusammenhang mit der Erstellung von Geometrieeobjekten ist folgender Link zu empfehlen:

[http://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy\\_slides2.pdf](http://www.gis.usu.edu/~chrisg/python/2009/lectures/ospy_slides2.pdf)

Als **Übung** soll das Skript dahingehend angepasst werden, dass in die neue Ebene alle Gemeinden von Solothurn geschrieben werden. Die Id-Spalte soll nachgeführt und der Name der Gemeinde übernommen werden. Das Bemerkungsfeld kann frei bleiben.

Als weitere **Übung** soll ein Python Skript geschrieben werden, welches die umgebenden Rechtecke (sog. MBR = minimum bounding rectangle) jeder Gemeinde als Polygone in einer separaten Datei einschliesslich des zugehörigen Namens speichert. Die Ein- und Ausgabedateien sollen als Parameter beim Aufruf des Skripts mitgegeben werden können (ebenso das Format der Ausgabedatei).

Ein möglicher Aufruf des Skripts wäre demnach:

```
python writeMBR.py Gemeinden_Solothurn.shp gemMBR.tab 'MapInfo File'
Name
```

Optional könnte noch ein Attributfilter mitgegeben werden können, dass nur die MBRs gewisser Gemeinden geschrieben werden sollen.

Die Lösung sähe zB wie folgt aus:

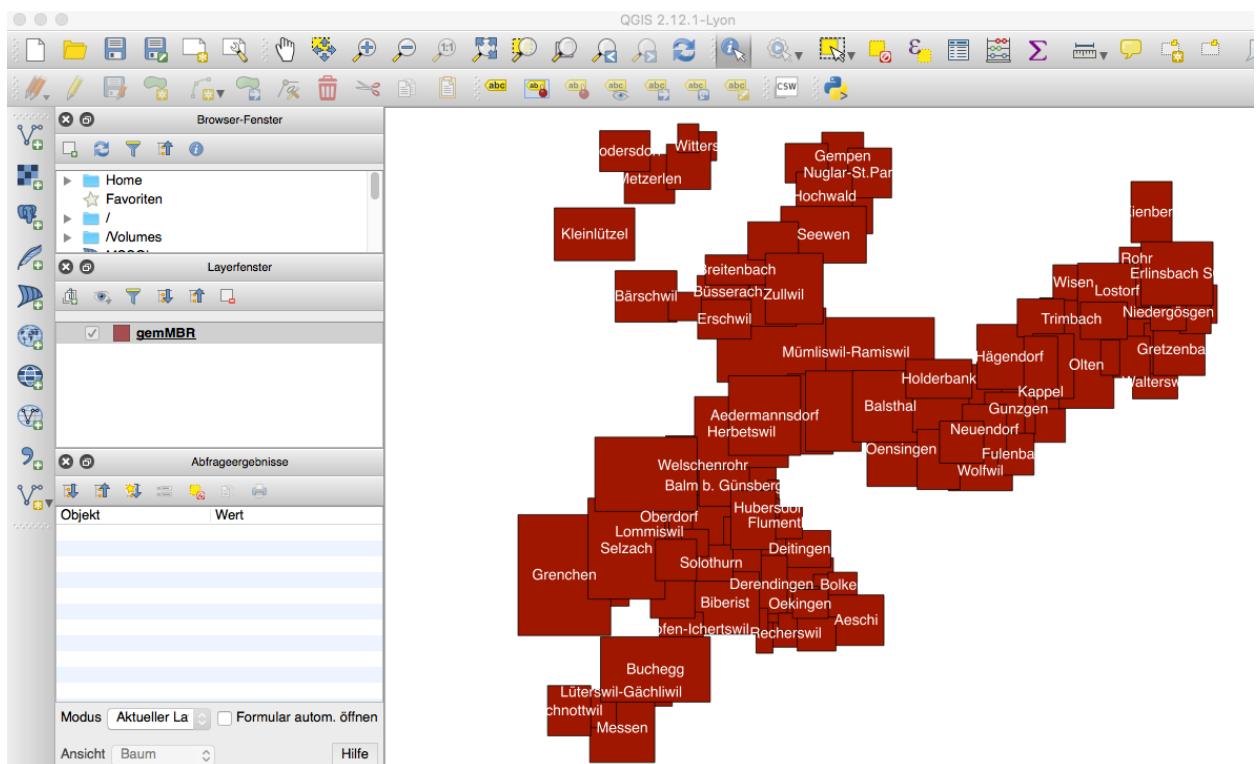


Abbildung 9 Umgebende Rechtecke der Gemeinden von Solothurn

## 6.6 Umprojektion einer Ebene durch Umprojektion jedes einzelnen Datensatzes

In diesem Beispiel werden die Erkenntnisse von einem vorigen Kapitel (Umprojektion) aufgefrischt und auf einen gesamten Datensatz angewendet: die Gemeinden von Solothurn werden einzeln umprojiziert und in einem neuen Datensatz gespeichert. Dieser Vorgang könnte gut mit dem Einbau eines Attributfilters kombiniert werden.

```
import sys
import os
import osr
import ogr

#Datensatz aufbereiten und SRS bestimmen:
#gemeinden SO in LV03
shapefileSO = ogr.Open('Daten/Gemeinden_Solothurn.shp')
if shapefileSO is None:
```

```

    print ("Datensatz konnte nicht geoeffnet werden.\n")
    sys.exit()
sogemeinden = shapefileSO.GetLayer(0)
srcProjection = osr.SpatialReference()
srcProjection.ImportFromWkt(sogemeinden.GetSpatialRef().ExportToWkt())

dstProjection = osr.SpatialReference()
dstProjection.ImportFromEPSG(4326)

#Transformationsparameter
transformdef = osr.CoordinateTransformation(srcProjection,
dstProjection)
driver = ogr.GetDriverByName('ESRI Shapefile')
if os.path.exists('Daten/GEMEINDEN_SO_WGS84.shp'):
    driver.DeleteDataSource('Daten/GEMEINDEN_SO_WGS84.shp')

#Zieldatensatz vorbereiten:
destinationFile =
driver.CreateDataSource('Daten/GEMEINDEN_SO_WGS84.shp')
destinationLayer =
destinationFile.CreateLayer('Daten/GEMEINDEN_SO_WGS84',
dstProjection, geom_type=ogr.wkbPolygon)

#Clone Attribute vom Ursprungs- in Ziellayer
for i in range(sogemeinden.GetLayerDefn().GetFieldCount()):

destinationLayer.CreateField(sogemeinden.GetLayerDefn().GetFieldDefn(i))

#Umprojektion pro Features/DS
gemsofeature = sogemeinden.GetNextFeature()

while gemsofeature:
    curgemgeometry = gemsofeature.GetGeometryRef()
    transformedgeometry = curgemgeometry.Clone()
    transformedgeometry.Transform(transformdef)
    gemsofeature.SetGeometry(transformedgeometry)
    destinationLayer.CreateFeature(gemsofeature)
    gemsofeature.Destroy()
    gemsofeature = sogemeinden.GetNextFeature()

shapefileSO.Destroy()
destinationFile.Destroy()
print ("*" * 50)
print ("Umprojektion fertig!")
print ("*" * 50)

print ("Ausgabe in Shapefile Daten/GEMEINDEN_SO_WGS84.shp")

```

## 6.7 Umprojektion von Rasterdaten

So wie Vektordaten durch Umprojektion von einem Bezugssystem in ein anderes transferiert werden können, können auf Kommandozeilen-Ebene auch Rasterdaten umprojiziert werden. Dazu dient der Befehl `gdalwarp()`, der im Kommandozeilenfenster abgesetzt wird. Ein Beispiel dazu sei folgender Befehl erwähnt:

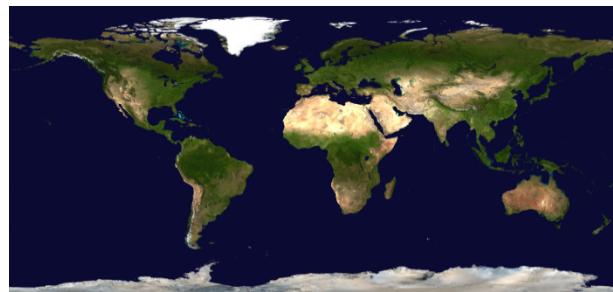
```
gdalwarp -t_srs "+proj=merc +datum=WGS84" geoworld.tif  
mercator.tif'
```

Dieser Befehl transformiert die Rasterdatei `geoworld.tif` in die Datei `mercator.tif`, welche in der Mercatorprojektion schliesslich vorliegt.

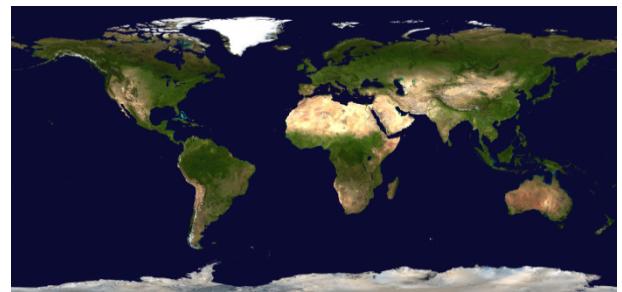
Ausgehend von der Datei `world.png` werden drei verschiedene Umprojektionsbeispiele vorgestellt (vgl. auch Abbildungen weiter unten):

```
import os  
  
os.system('gdal_translate -a_srs WGS84 -a_ullr -180 90 180 -90  
world.png geoworld.tif')  
os.system('gdalwarp -t_srs "+proj=merc +datum=WGS84"  
geoworld.tif mercator.tif')  
os.system('gdalwarp -t_srs "+proj=ortho +datum=WGS84"  
geoworld.tif ortho.tif')
```

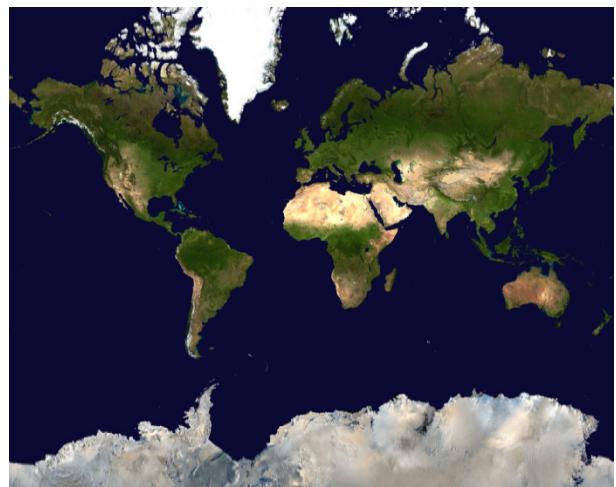
Als Ergebnis zeigen sich folgende Bilder:



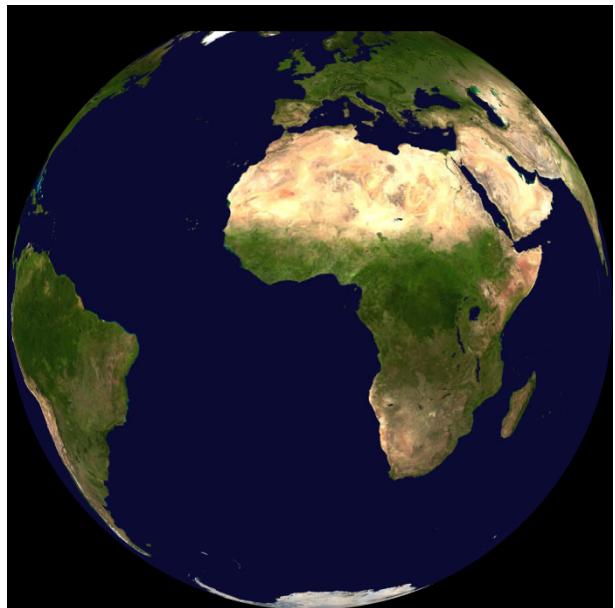
World.png (Ausgangsdaten)



Geoworld.tif



Mercator.tif



Ortho.tif

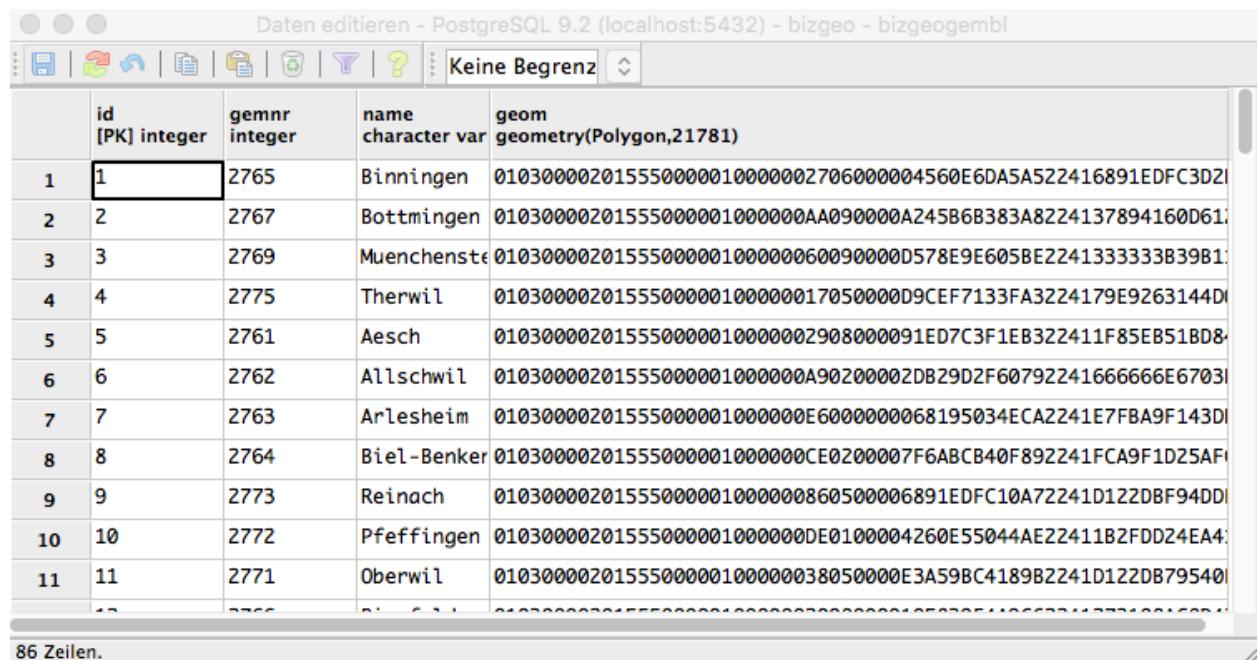
## 7. Arbeiten mit PostgreSQL/PostGIS

Mit Python kann die Datenbank PostgreSQL direkt angesprochen werden über das Modul psycopg2. Damit können in Python direkt SQL Befehle auf die Datenbank abgesetzt und ggf. auch verarbeitet werden.

### 7.1 Laden von Beispieldaten nach PostgreSQL/PostGIS

Über die Bedieneroberfläche pgAdminIII können Daten nach PostgreSQL/PostGIS importiert werden. Eine Beispieldatei mit Gemeindedaten vom Kanton Baselland wird in diesem Beispiel verwendet. Dazu ist die Datei `_gemb1_export` mittels pgAdminIII zu importieren.

Die Daten lassen sich anschliessend sowohl in pgAdminIII als auch QGIS darstellen:



The screenshot shows a pgAdmin III interface with a title bar "Daten editieren - PostgreSQL 9.2 (localhost:5432) - bizgeo - bizgeogemb1". Below the title bar is a toolbar with icons for file operations like New, Open, Save, and a search bar labeled "Keine Begrenz". The main area displays a table with four columns: "id [PK] integer", "gemnr integer", "name character var", and "geom geometry(Polygon,21781)". The table contains 11 rows, each representing a community. The first row has the id 1 highlighted. The last row is partially visible. At the bottom left of the table area, it says "86 Zeilen."

	<b>id [PK] integer</b>	<b>gemnr integer</b>	<b>name character var</b>	<b>geom geometry(Polygon,21781)</b>
1	1	2765	Binningen	01030000201555000001000002706000004560E6DA5A522416891EDFC3D21
2	2	2767	Bottmingen	0103000020155500000100000AA090000A245B6B383A8224137894160D61
3	3	2769	Muenchensteig	0103000020155500000100000600900000D578E9E605BE2241333333B39B1
4	4	2775	Therwil	010300002015550000010000017050000D9CEF7133FA3224179E9263144D
5	5	2761	Aesch	01030000201555000001000002908000091ED7C3F1EB322411F85EB51BD8
6	6	2762	Allschwil	0103000020155500000100000A90200002DB29D2F60792241666666E6703
7	7	2763	Arlesheim	0103000020155500000100000E6000000068195034ECA2241E7FBA9F143D
8	8	2764	Biel-Benken	0103000020155500000100000CE0200007F6ABCB40F892241FCA9F1D25AF
9	9	2773	Reinach	0103000020155500000100000860500006891EDFC10A72241D122DBF94DD
10	10	2772	Pfeffingen	0103000020155500000100000DE010004260E55044AE22411B2FDD24EA4
11	11	2771	Oberwil	010300002015550000010000038050000E3A59BC4189B2241D122DB79540
				010300002015550000010000038050000E3A59BC4189B2241D122DB79540

Abbildung 10 Gemeindedaten in PostgreSQL

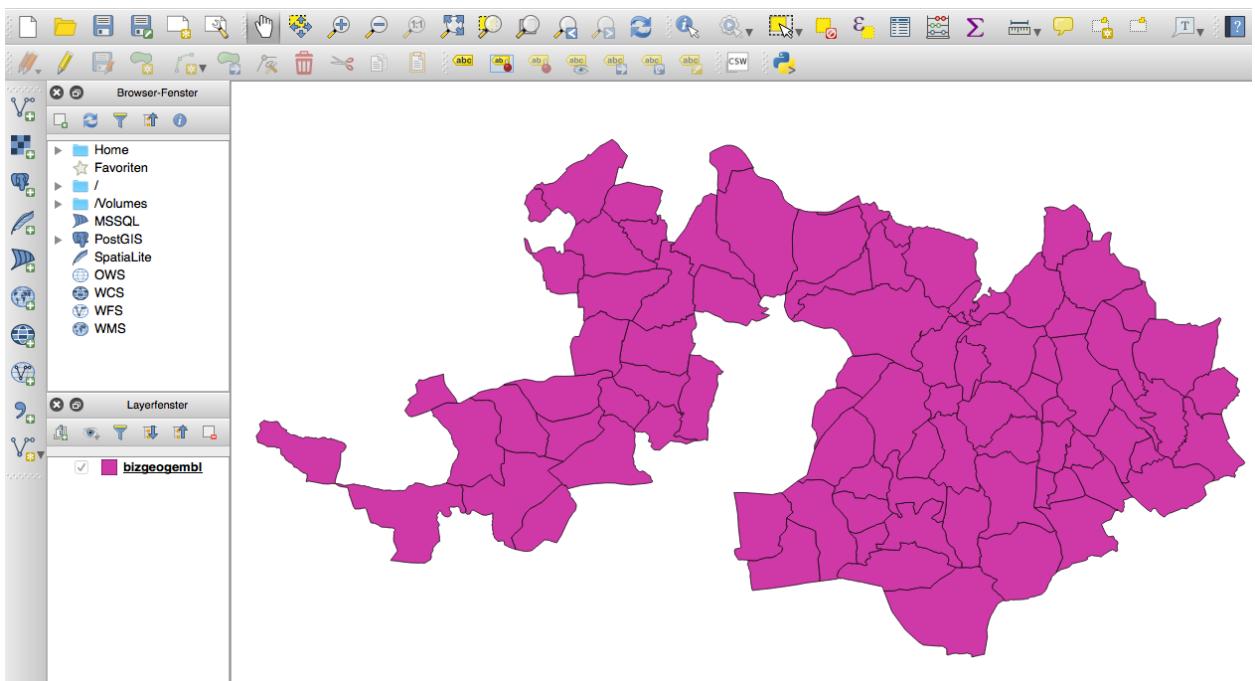


Abbildung 11 Gemeindedaten in QGIS

Die Daten stehen nur zur Abfrage mittels Python bereit.

## 7.2 Verbindung zur Datenbank

Die Verbindung zur PostgreSQL-Datenbank geschieht wie folgt:

```
import psycopg2

#Connect to PostgreSQL
connection = psycopg2.connect(dbname="bizgeo",
host="localhost", user="postgres", password="postgres",
port="5432")
cursor = connection.cursor()
```

Es wird also ein sog. `cursor`-Objekt benötigt, welches abhängig von einem `connection`-Objekt in der Lage ist, die gewünschten Transaktionen via SQL auszuführen.

Ein SQL Befehl wird anschliessend mit folgender Syntax abgesetzt:

```
sqlcommand = "Select * from bizgeogembl;"
cursor.execute(sqlcommand)
```

Das Ergebnis, das durch einen SQL-Befehl zurückgegeben wird, kann anschliessend wie folgt ausgewertet werden:

```
cursor.execute(<SQL-Befehl>
<Auswertung des cursor-Objektes>
```

Damit ist die Grundstruktur für das Arbeiten mit PostgreSQL gegeben. Wenn räumliche Daten verarbeitet werden, muss in PostgreSQL der Zusatz PostGIS installiert sein. Dank PostGIS können räumliche Daten in räumlichen Datentypen gespeichert und mit entsprechenden räumlichen Methoden bearbeitet und abgefragt werden.

### 7.3 Abfrage - SQL Select Statement

Wie bereits weiter oben gesehen, kann auf eine Datenbank direkt eine Abfrage ausgeführt und anschliessend ausgewertet werden. Dazu muss zuerst die Verbindung zur Datenbank hergestellt worden sein. Dann kann die Abfrage formuliert und abgesetzt werden. Das Resultat kann anschliessend in Python weiterverarbeitet werden:

```
#Query Database
sqlstring = "SELECT name, gemnr FROM bizgeogembl;"
cursor.execute(sqlstring)

for name, gemnr in cursor:
    print ("Gem: %s, Nr: %s" %(name, gemnr))
```

Nebst Abfragen mit einem Select Befehl können natürlich auch weitere SQL Befehle abgesetzt werden wie bspw. Insert oder Update etc.

### 7.4 Weitere Abfragen

Zur Übung können nun weitere Abfragen ausgeführt und in Python verarbeitet werden:

Ermittle alle Gemeinden von Baselland, welche mit dem Buchstaben 'S' beginnen.

Ermittle alle Gemeinden von Baselland, welche eine Fläche haben, die grösser ist als 6km<sup>2</sup>. Ermittle auch die Anzahl der erhaltenen Datensätze.

Speichere die alle Gemeinde-Datensätze als Esri Shape-Daten ab.

## 7.5 Import von Daten mittels Python

Im folgenden Teil werden die bereits mehrfach behandelten Gemeindedaten von Solothurn, die als Esri Shapen Datei vorliegen in die Datenbank PostgreSQL importiert:

```
# -*- coding: utf-8 -*-
"""
Created on Thu Sept 2 2016

@author: Hans-Jörg Stark
"""

import psycopg2
import sys
import time
import ogr
startTime = time.time()

#Verbindung zur PostgreSQL DB
connection = psycopg2.connect(dbname="bizgeo",
host="localhost", user="postgres", password="postgres",
port="5432")
cursor = connection.cursor()

#Tabelle erstellen inkl. notwendiger Indices
cursor.execute("""CREATE TABLE gemso (
    id      SERIAL,
    gemnr   INTEGER,
    beznr   INTEGER,
    name    CHARACTER VARYING(100),
    PRIMARY KEY (id)
    """
)
cursor.execute("CREATE INDEX gemnrIndex ON gemso(Gemnr)")
cursor.execute("SELECT AddGeometryColumn('gemso', 'geom',
21781, 'MULTIPOLYGON', 2)")
cursor.execute("CREATE INDEX geomIndex ON gemso USING GIST
(geom)")
#WICHTIG: speichern der Transaktion!
connection.commit()

#####
#Import aller Datensaetze der Gemeinden von Solothurn
shapefile = ogr.Open("Daten/Gemeinden_Solothurn.shp")
if shapefile is None:
    print ("Datensatz konnte nicht geoeffnet werden.\n")
    sys.exit()

layer = shapefile.GetLayer(0)

#Extrahiere Gemeindegeometrie
gemgeometry = None
```

```

for feature in layer:
    #Extrahiere Gemeinde-Name
    gemname = feature.GetField("gmde_name")
    print ("Gemeinde %s in Datenbank eingetragen." %gemname)
    #Extrahiere Gemeinde- und Bezirks-Nummer
    gemnr = int(feature.GetField("gmde_nr"))
    beznr = int(feature.GetField("bzrk_nr"))
    #Extrahiere Geometrie Polygon
    gemgeometry = feature.GetGeometryRef()
    gemgeometryaswkt = gemgeometry.ExportToWkt()
    sqlstring = "INSERT INTO gemso (name, gemnr, beznr, geom)
VALUES ('%s', %s, %s, ST_GeomFromText('%s', 21781))" %
(gemname,gemnr,beznr,gemgeometryaswkt)
    print (sqlstring)
    cursor.execute(sqlstring)
    connection.commit()
endTime = time.time()
print ("Took %0.4f seconds" % (endTime-startTime))

```

## 8. Aufruf und Verarbeiten von Geowebdiensten

### 8.1 Grundlagen zu Geowebdiensten

Geowebdienste wie WMS (WebMapService) und WFS (WebFeatureService) werden heute bereits in vielen Geodateninfrastrukturen regelmässig eingesetzt. Vor allem der WMS hat sich als Darstellungsdienst in der Praxis bewährt. Der WFS wird nicht ganz so häufig eingesetzt. Dies zu einen, weil die angeforderten Vektordaten jedem Client zur Verfügung stehen und damit im Gegensatz zum WMS ggf. die Originaldaten veröffentlicht werden. Zum andern kann je nach Umfang des Datensatzes die Auslieferung, dh. der Datenstrom, der ausgeliefert wird, sehr umfangreich sein und damit eine starke Belastung für die Infrastruktur darstellen - sei dies seitens des Datenanbieters (Server), sei dies seitens der Datenbezüger (Clients).

In der Schweiz gibt es den eCH Standard "eCH-0056 Anwendungsprofil Geodienste". Darin werden gewisse Richtlinien im Umgang mit Geowebdiensten geregelt. U.a. findet sich dort auch eine knappe Zusammenfassung der Funktionsweise und der Idee hinter Geowebdiensten. An dieser Stelle sei auf dieses Dokument verwiesen. Darin wird bspw. die grundlegende Funktionsweise von Geowebdiensten beschrieben:

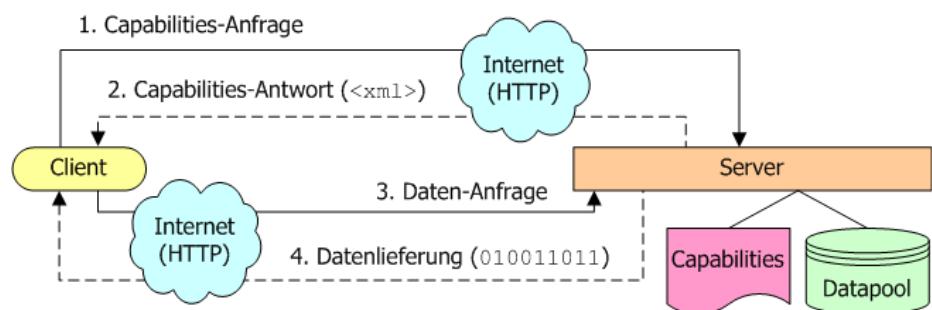


Abbildung 12 Funktionsweise von Geowebdiensten (Quelle: eCH-0056)

Geowebdienste wie WMS und WFS basieren alle auf demselben Funktionsschema: Ein Client kommuniziert mit einem Server (über das http Protokoll). Die Spezifikationen zu den erwähnten Geowebdiensten definieren lediglich die Kommunikation (Wie muss die Frage, wie die Antwort aussehen?) zwischen Web Service Bezüger (Client) und Web Service Anbieter (Server). Sie machen keine Vorgaben dazu, wie eine Anfrage abgearbeitet wird (vgl. Abbildung 12).

1. Client kontaktiert Server und fordert Capabilities-Dokument an
2. Server liefert XML-formatierte Capabilities vom gewünschten Service an den Client
3. Client fordert Daten vom Server an
4. Server liefert die angeforderten Daten im verlangten Format

Die oben erwähnten und in Abbildung 12 dargestellten vier Schritte bilden die Grundfunktionalität eines Geowebdienstes, wie es WMS und WFS sind. Je nach Geowebdienst sind weitere Interaktionen zwischen Client und Server möglich, beispielsweise das Abfragen weiterer Informationen zu einem Objekt, einer Kartenebene etc. Wichtig ist, dass über das sog. Capabilities Dokument der Dienst sich selbst beschreibt und daher sowohl für eine Kommunikation Mensch-Maschine als auch Maschine-Maschine einsetzbar ist.

## 8.2 Ansprechen von WMS und WFS mittels Python

### 8.2.1 WMS

Als erstes Beispiel wird eine sog. GetMap() Anfrage mittels Python an einen Server gestellt. Dabei wird die GetMap() Anfrage ausgeführt. Das erhaltene Resultat - eine Rasterdatei - wird in einer neuen Datei lokal abgespeichert. Diese steht anschliessend zur weiteren Verarbeitung - bspw. mittels GDAL - zur Verfügung.

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

import os, shutil, sys
import urllib.request
import gdal
from gdalconst import *

def download(url, dest, fileName=None):
    #based on:
    #http://stackoverflow.com/questions/862173/how-to-download-a-
    #file-using-python-in-a-smarter-way/863017#863017

    print("*****")
    print(url)
    print("*****")
    r= urllib.request.urlopen(url)

    fileName = os.path.join(dest, fileName)
    with open(fileName, 'wb') as f:
        shutil.copyfileobj(r,f)
    r.close()

if __name__=='__main__':
```

```

wmsfile = "sogis.gif"
if os.path.exists(wmsfile):
    os.remove(wmsfile)

path2save2 = "" #Zielpfad
wmslink =
"http://geoweb.so.ch/wms/sogis_natgef.wms?service=wms&VERSION=1
.3.0&REQUEST=GetMap&LAYERS=wassgef_01,steinschlag_01,rutsch_01&
STYLES=&CRS=EPSG:21781&BBOX=605000,225000,612500,230000&WIDTH=7
20&HEIGHT=480&FORMAT=image/png"

download(wmslink,path2save2,wmsfile)

#Open Rasterfile
fn = os.path.join(path2save2,wmsfile)
ds = gdal.Open(fn, GA_ReadOnly)
if ds is None:
    print('Could not open ' + fn)
    sys.exit(1)

#check with gdalinfo
os.system('gdalinfo %s' %fn)

```

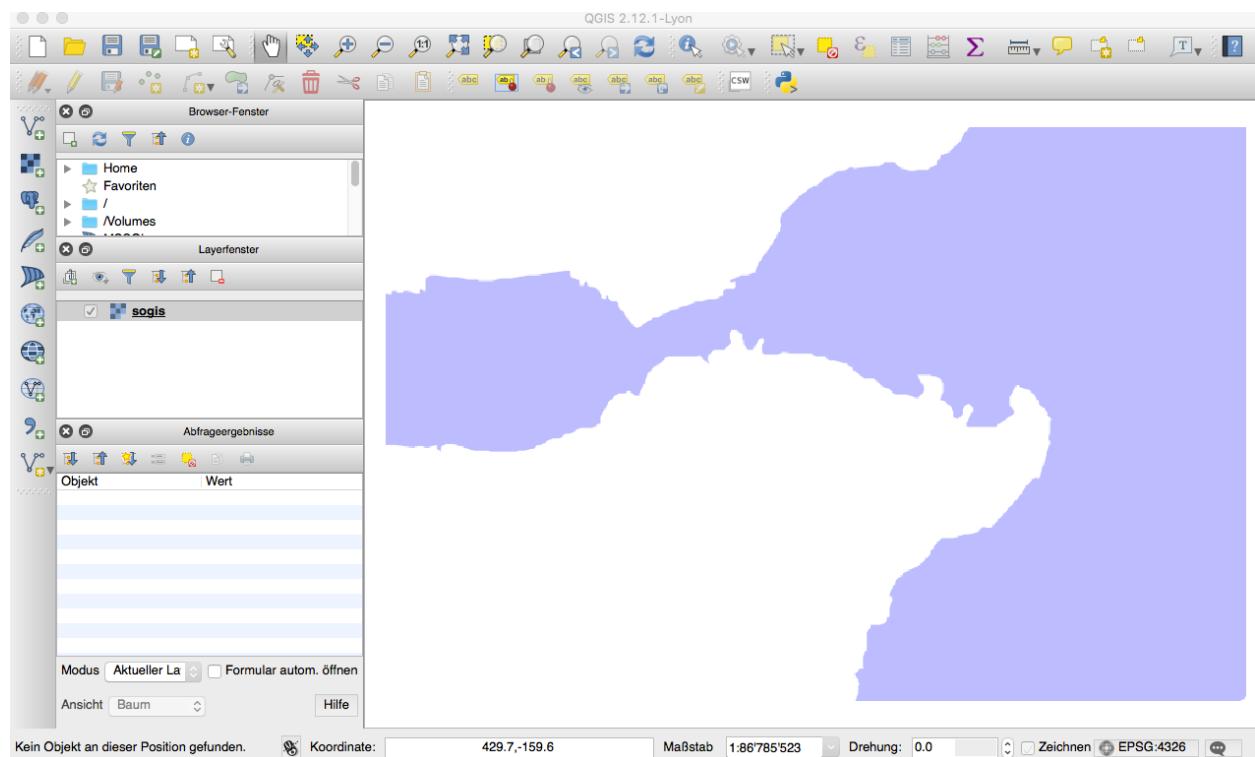


Abbildung 13 WMS Daten von Solothurn visualisiert in QGIS

### 8.2.2 WFS

Die Abfrage bzw. das Ausführen einer WFS Anfrage verläuft grundsätzlich gleich wie die WMS Anfrage. Allerdings ist das erhaltene Resultat hinsichtlich Format ein anderes: statt Rasterdaten werden Vektordaten zurückgeliefert. Entsprechend anders ist das Resultat zu behandeln.

Im folgenden Beispiel werden Vektordaten bezogen und anschliessend mit dem Befehl ogr2ogr in eine Shape-Datei konvertiert um sie bspw. im Anschluss in QGIS betrachten und weiter verarbeiten zu können.

```
# -*- coding: utf-8 -*-
import os, sys, shutil
import urllib.request
import ogr
import gdal
from gdalconst import *

#WFS Daten von:
#http://www.are.zh.ch/internet/baudirektion/are/de/geoinformation/geodienste_uebersicht/WebFeatureService.html

def download(url, dest, fileName=None):
    #based on:
    #http://stackoverflow.com/questions/862173/how-to-download-a-
    #file-using-python-in-a-smarter-way/863017#863017
    print("*****")
    print ("Start downloading of %s" %url)
    print("*****")

    r= urllib.request.urlopen(url)

    try:
        fileName = os.path.join(dest, fileName)
        with open(fileName, 'wb') as f:
            shutil.copyfileobj(r,f)
        print ("Saved in %s" %fileName)
    finally:
        r.close()

def convert2shp(path2save2,wfsfile,outputshapefile):
    fn = os.path.join(path2save2,wfsfile)

    driver = ogr.GetDriverByName('ESRI Shapefile')
    if os.path.exists(outputshapefile):
        driver.DeleteDataSource(outputshapefile)

    #convert GMLfile to shape - if needed...
```

```

ogr2ogrstring = 'ogr2ogr -f "ESRI Shapefile" %s %s'
%(outputshapefile,fn)
print (ogr2ogrstring)
os.system(ogr2ogrstring)
print ("Conversion successful...")

#... oder GML...
wfsfile = ogr.Open(fn)
if wfsfile is None:
    print ("Datensatz konnte nicht geoeffnet werden.\n")
    sys.exit( 1 )

layer = wfsfile.GetLayer(0)
lname = layer.GetName()

print ("Layername: ", lname)

#Print out number of records:
numftrs = layer.GetFeatureCount()
print ("Anzahl Features in GML Datei: %d" %numftrs)
print ("")

print ("Count Field Count",
layer.GetLayerDefn().GetFieldCount())
for feat in range(numftrs):
    for i in range(layer.GetLayerDefn().GetFieldCount()):
        field_defn = layer.GetLayerDefn().GetFieldDefn(i)
    try:
        print (" %s: %s" %(field_defn.GetName(),
layer.GetFeature(feat).GetField(i)))
    except:
        pass
    print()

#Get Extent
extent = layer.GetExtent()
print ("Ausdehnung:", extent)
print ("Oben-links:", extent[0], extent[3])
print ("Unten-rechts:", extent[1], extent[2])

if __name__=='__main__':
    ######
    #Punktdaten:
    wfsfile = "testpoints.gml"
    outputshapefile = 'wfstestpoints.shp'
    path2save2 = ""
    wfsurl =
    "http://maps.zh.ch/wfs/HaltestellenZHWF

```

```

=1.1.0&Request=getfeature&TYPENAME=haltestellen&MAXFEATURES=
30"
download(wfsurl,path2save2 ,wfsfile)
convert2shp(path2save2,wfsfile,outputshapefile)

#Liniendaten:
wfsfile = "testlines.gml"
outputshapefile = 'wfstestlines.shp'
path2save2 = ""
wfsurl =
"http://maps.zh.ch/wfs/GemZHWFSSERVICE=WFS&VERSION=1.1.0&Re
quest=getfeature&TYPENAME=grenzen&MAXFEATURES=100"

download(wfsurl,path2save2 ,wfsfile)
convert2shp(path2save2,wfsfile,outputshapefile)

```

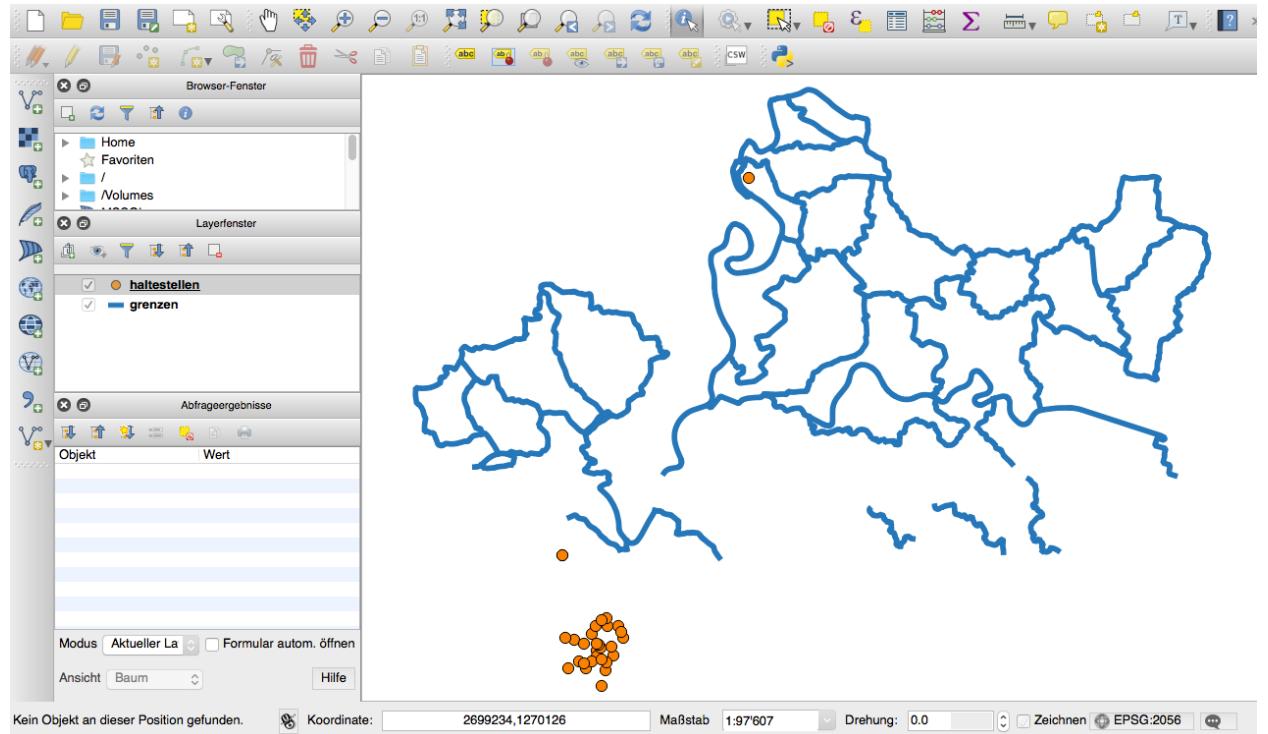


Abbildung 14 WFS Daten in QGIS visualisiert

## 9. Arbeiten mit der Bibliothek Shapely

Die Bibliothek Shapely ist umfangreich und verdiente daher ein eigenes Kapitel. Aus Ressourcengründen wird an dieser Stelle aber lediglich kurz darauf eingegangen. Mit shapely können Geodaten in Form von Punkten, Linien, Polygonen, Multipoints usf. verarbeitet werden. Das Modul muss für dessen Verwendung zuerst installiert werden!

Eine gute Einführung ist unter <http://toblerity.org/shapely/manual.html> zu finden.

Im Folgenden werden zwei Beispiele erwähnt. Als erstes wird gezeigt, wie die Extraktion von Zentroid und Flächenmass in Shapely geschehen kann. Es werden die Zentroide und das Flächenmass der Gemeinden ausgegeben:

```
import ogr
import shapely.wkt

shapefile = ogr.Open("Daten/Gemeinden_Solothurn.shp")
if shapefile is None:
    print ("Datensatz konnte nicht geoeffnet werden.\n")
    sys.exit()

layer = shapefile.GetLayer(0)

#Gemeindegeometry extrahieren:
geometry = None
for feature in layer:
    #Extract Gemeinde-Name
    gemname = feature.GetField("gmde_name")
    #Get Geometry (Polygon)
    gemgeometry = feature.GetGeometryRef()
    #"Convert" Geometry to shapely-geometry
    gemgeomaswkt = gemgeometry.ExportToWkt()
    shapelypolygon = shapely.wkt.loads(gemgeomaswkt)
    #Extract Centroid
    centroid_point = shapelypolygon.centroid
    x=centroid_point.x
    y=centroid_point.y
    area = shapelypolygon.area
    #Printout Information
    print ("Gemeinde %s hat folgenden Zentroid: (%f, %f) und folgende Flaeche %fm2" %(gemname, x, y, area))
```

Im nächsten Beispiel wird die Länge des gemeinsamen Grenzverlaufs der Gemeinden Oensingen und Kestenholz (SO) ermittelt. Das Ergebnis ist eine Linie, dh. der Grenzverlauf, den beide Gemeinden gemeinsam haben. Als Ergebnis werden sowohl die Länge der Linie, der Geometriertyp, die Anzahl der Liniensegmente als auch die einzelnen Stützpunkte ausgegeben:

```

import ogr
import shapely.wkt
import sys
from numpy import array
from pprint import pprint
from shapely.geometry import LineString

shapefile = ogr.Open("Daten/Gemeinden_Solothurn.shp")
if shapefile is None:
    print ("Der Datensatz konnte nicht geöffnet werden.\n")
    sys.exit()

layer = shapefile.GetLayer(0)

#Gemeindegeometrie extrahieren:
geometry = None
for feature in layer:
    #Extract Geometries for Muttenz and Pratteln
    if feature.GetField("gmde_name") == 'Oensingen':
        geomOensingen = feature.GetGeometryRef()
        Oensingengeomaswkt = geomOensingen.ExportToWkt()
        shapelypolygonOensingen =
shapely.wkt.loads(Oensingengeomaswkt)
    elif feature.GetField("gmde_name") == 'Kestenholz':
        geomKestenholz = feature.GetGeometryRef()
        Kestenholzgeommaswkt = geomKestenholz.ExportToWkt()
        shapelypolygonKestenholz =
shapely.wkt.loads(Kestenholzgeommaswkt)

#compute intersection
intersectionline =
shapelypolygonOensingen.intersection(shapelypolygonKestenholz)
type = intersectionline.geom_type
vertices = len(intersectionline.geoms)
print ("")
print ("Laenge des gemeinsamen Grenzverlaufs (vom Typ %s): %fm
mit %i Liniensegmenten" %(type, intersectionline.length,
vertices))
print ("")

#Extraktion der Haltepunkte der Schnittlinie
i=0
for vertex in intersectionline.geoms:

```

```

i=i+1
x1 = vertex.coords[0][0] # 1. Punkt der Linie, X-Koordinate
y1 = vertex.coords[0][1] # 1. Punkt der Linie, Y-Koordinate

print ("Stützpunkt[%i]: (x=%f, y=%f)" %(i,x1,y1))
if i==len(intersectionline.geoms):
    x1=vertex.coords[1][0] # 2. Punkt der Linie, X-
Koordinate
    y1=vertex.coords[1][1] # 2. Punkt der Linie, Y-
Koordinate
    print ("Stützpunkt[%i]: (x=%f, y=%f)" %(i+1,x1,y1))

```

Die gemeinsame Grenze sieht visualisiert im Kontext der beiden Gemeinden wie folgt aus:

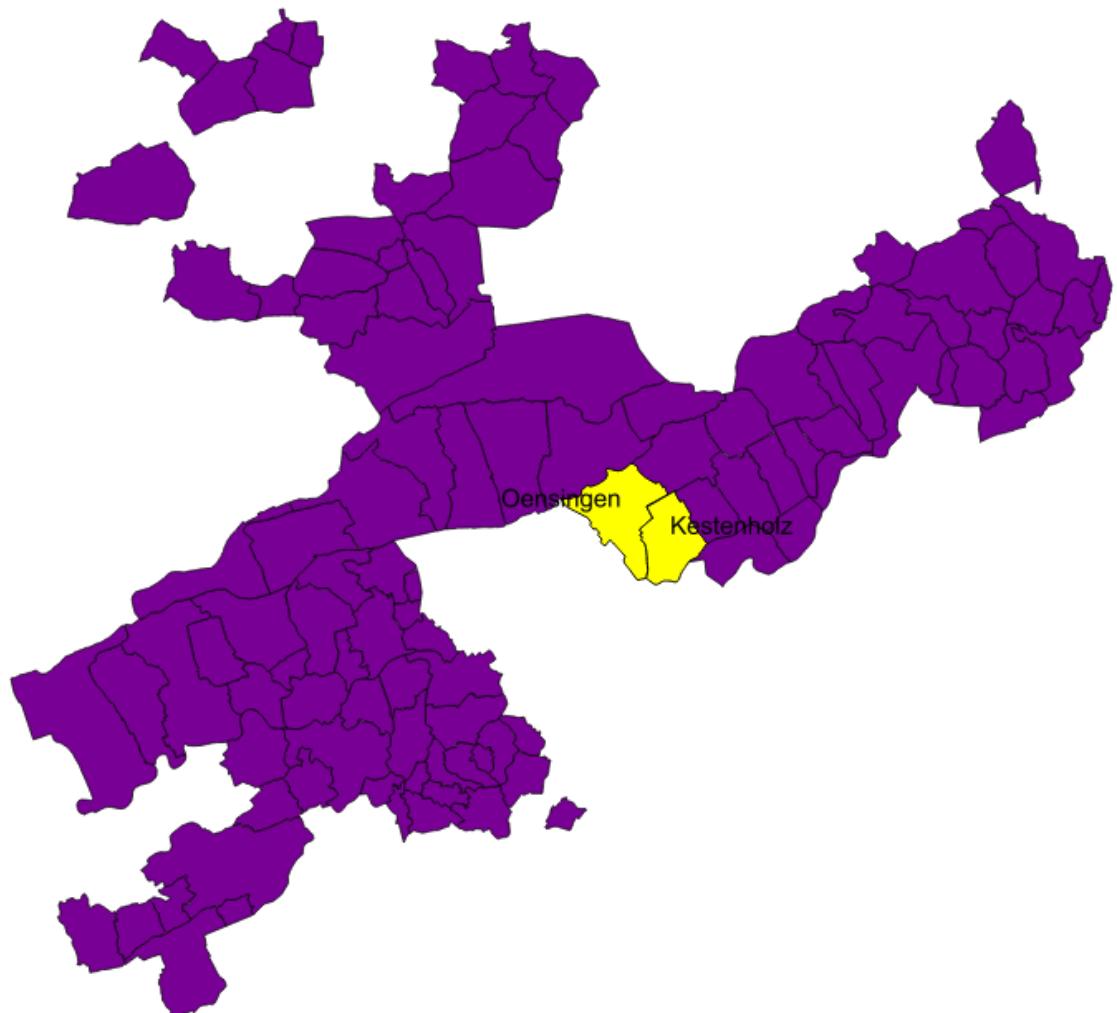


Abbildung 15 Suche des gemeinsamen Grenzverlaufs von Oensingen und Kestenholz im Kanton Solothurn

## 10. Arbeiten mit der Bibliothek Fiona

Fiona ist eine weitere Bibliothek für die Verarbeitung von Vektordaten. Im Folgenden werden die einfachsten und gängigsten Funktionen für die Erstellung und das Lesen von Geometriedaten vorgestellt. Eine ausführliche Quelle der Dokumentation ist unter <http://toblerity.org/fiona/manual.html> zu finden. Sean Gilles schreibt auf dieser Seite u.a.:

*"Please understand this: Fiona is designed to excel in a certain range of tasks and is less optimal in others. Fiona trades memory and speed for simplicity and reliability. Where OGR's Python bindings (for example) use C pointers, **Fiona copies vector data from the data source to Python objects**. These are simpler and safer to use, but more memory intensive. **Fiona's performance is relatively more slow if you only need access to a single record field** – and of course if you just want to reproject or filter data files, nothing beats the ogr2ogr program – but **Fiona's performance is much better than OGR's Python bindings if you want all fields and coordinates of a record**. The copying is a constraint, but it simplifies programs. With Fiona, you don't have to track references to C objects to avoid crashes, and you can work with vector data using familiar Python mapping accessors. Less worry, less time spent reading API documentation. [...]*

*In what cases would you **benefit** from using Fiona?*

- *If the features of interest are from or destined for a file in a non-text format like ESRI Shapefiles, Mapinfo TAB files, etc.*
- *If you're more interested in the **values of many feature properties than in a single property's value**.*
- *If you're more interested in all the coordinate values of a feature's geometry than in a single value.*
- *If your processing system is distributed or not contained to a single process.*

*In what cases would you **not benefit** from using Fiona?*

- *If your data is in or destined for a JSON document you should use Python's json or simplejson modules.*
- *If your data is in a RDBMS like PostGIS, use a Python DB package or ORM like SQLAlchemy or GeoAlchemy. Maybe you're using GeoDjango already. If so, carry on.*

- If your data is served via HTTP from CouchDB or CartoDB, etc, use an HTTP package (`httpclient2`, `Requests`, etc) or the provider's Python API.
- If you can use `ogr2ogr`, do so.

[..]"

Hier kommt ein erstes einfaches Beispiel zum Lesen einer Vektordatei und davon ein einzelner Datensatz:

```
import fiona
import rasterio
import pprint

#c = fiona.open('_TestETH/Gemeinden_Solothurn.shp', 'r')

print("Anzahl Datensätze: %i " %len(list(c)))

print("Format: %s" %c.driver)

print("Geo-Referenzsystem: %s" %c.crs)

print("Ausdehnung: %s" %str(c.bounds))

#Ausgabe der Sachdaten und Geometrie von Datensatz 1
with c as src:
    #Geometrie-Typ
    pprint.pprint(src[1]['geometry']['type'])

    #Saemtliche Informationen
    pprint.pprint(src[1])
```

Das nächste Beispiel ist etwas ausführlicher und liest zusätzlich noch die Koordinaten aus:

```
import fiona
from fiona.crs import to_string
import sys

with fiona.open('_TestETH/GemSoPnts.shp', 'r') as source:
    bounds = str(source.bounds)
    print("MBR Dataset: %s" %bounds)
    crs = to_string(source.crs)
    print("Coordinate Reference System: %s" %crs)
    driver = source.driver
    print("Fileformat: %s" %driver)

    cnt = 1
    for f in source:
```

```

if cnt < 2:
    curid = int(f['id'])
    print("Dataset ID: %i" %curid)

    tabschema=source.schema.keys()
    print("Table-Schema: %s" %tabschema)
    colnames=source.schema['properties'].keys()
    print("Columnnames: %s" %colnames)
    #get all Columnnames:
    colnamesandvalues = f['properties']
    print("Columnnames and -values: %s"
          %colnamesandvalues)
    for colname in f['properties']:
        colval = f['properties'][colname]
        print("%s: %s" %(colname,colval))

geom = f['geometry']
geomtype = geom['type']
print("Entire Geometry of type %s:" %geomtype)

print()
if geomtype == "Polygon":
    rings = geom['coordinates']
    numpnts = len(rings[0])
    print("Number of Coordinate-Pairs: %i"
          %numpnts)
    print()
    for coord in rings[0]:
        x=coord[0]
        y=coord[1]
        print("X: %f | Y: %f" %(x,y))

elif geomtype == "Point":
    coordinates = geom['coordinates']
    x=coordinates[0]
    y=coordinates[1]
    print("X: %f | Y: %f" %(x,y))
    cnt += 1
else:
    break

source.close()

```

Im nächsten Beispiel wird zusätzlich zu Fiona die Bibliothek Pyproj eingesetzt, um Daten zu schreiben. Es werden die Gemeinden von Solothurn in eine neue, eigene Esri-Shape-Datei geschrieben (das Logging-Modul hält ggf. auftretende Fehler fest):

```
import logging
import sys

from pyproj import Proj, transform

import fiona
from fiona.crs import from_epsg, to_string

logging.basicConfig(stream=sys.stderr, level=logging.INFO)

with fiona.open('_TestETH/Gemeinden_Solothurn.shp', 'r') as source:

    gem_schema = source.schema.copy()
    p_in = Proj(source.crs)

    with fiona.open(
        '_TestETH/GemSowith-pyproj.shp', 'w',
        crs=from_epsg(21781),
        driver=source.driver,
        schema=gem_schema,
    ) as gem:

        p_out = Proj(gem.crs)

        for f in source:
            try:
                if f['geometry']['type'] == "Polygon":
                    new_coords = []
                    for ring in f['geometry']['coordinates']:
                        x2, y2 = transform(p_in, p_out,
                                           *zip(*ring))
                        new_coords.append(zip(x2, y2))
                    f['geometry']['coordinates'] = new_coords
                    gem.write(f)

            except Exception as e:
                # Writing uncleanable features to a different shapefile
                # is another option.
                logging.exception("Error transforming feature %s:", f['id'])
```

Desgleichen kann auch Shapely eingesetzt werden in Kombination mit Fiona, um vektorielle Geodaten zu schreiben. Das folgende Beispiel gibt mögliche Lösungen zum Schreiben von Punkt-, Linien- und Polygonegeometrien wieder:

```

import fiona
# Shapely wird für die Definition der Geometrie benötigt
from shapely.geometry import Point, LineString, Polygon, mapping

#####
# Punktlayer erstellen:
# Schema: einfaches Dictionary mit Geometrie und Properties als keys
schema_pnt = {'geometry': 'Point', 'properties': {'Id': 'int', 'Name': 'str'}}

# Ein paar Punktgeometrien
points = [Point(272830.63, 155125.73), Point(273770.32, 155467.75), Point(273536.47, 155914.07), Point(272033.12, 152265.71)]

with fiona.open('_TestETH/mypointshapes.shp', 'w', 'ESRI Shapefile', schema_pnt) as pntlayer:
    cnt = 0
    for pnt in points:
        cnt += 1
        # Schema befüllen
        elem = {}
        # Geometrie wird mit der mapping function von shapely erstellt
        elem['geometry'] = mapping(pnt)
        # Attributwerte
        elem['properties'] = {'Name': 'Punkt ' + str(cnt), 'Id': cnt}
    # Erstellen des neuen Datensatzes / Records
    pntlayer.write(elem)

#####
# Linienlayer erstellen:
# Schema: einfaches Dictionary mit Geometrie und Properties als keys
schema = {'geometry': 'LineString', 'properties': {'Id': 'int', 'Name': 'str'}}

# Zwei einfache Liniengeometrien
lines = [LineString([(272830.63, 155125.73), (273770.32, 155467.75)]), LineString([(273536.47, 155914.07), (272033.12, 152265.71)])]
with fiona.open('_TestETH/mylineshapes.shp', 'w', 'ESRI Shapefile', schema) as layer:

```

```

cnt = 0
for line in lines:
    cnt += 1
    # Schema befüllen
    elem = {}
    # Geometrie wird mit der mapping function von shapely
erstellt
    elem['geometry'] = mapping(line)
    # Attributwerte
    elem['properties'] = {'Name': 'Linie ' + str(cnt), 'Id'
: cnt}
    # Erstellen des neuen Datensatzes / Records
    layer.write(elem)

#####
# Polygonlayer erstellen:
# Schema: einfaches Dictionary mit Geometrie und Properties als
keys
schema = {'geometry': 'Polygon', 'properties': {'Id': 'int',
'Name': 'str'}}

# Zwei einfache Polygongeometrien
polygons =
[Polygon([(272830.63,155125.73),(273536.47,155914.07),(273770.3
2,155467.75),(272033.12,152265.71)]),

Polygon([(270800.0,155100.0),(270800.0,155200.0),(271000.0,1552
00.0),(271000.0,155100.0)]),
polygon = Polygon([(0, 0), (1, 1), (1, 0)])
with fiona.open('_TestETH/mypolygonshapes.shp', 'w', 'ESRI
Shapefile', schema) as layer:
    cnt = 0
    for poly in polygons:
        cnt += 1
        # Schema befüllen
        elem = {}
        # Geometrie wird mit der mapping function von shapely
erstellt
        elem['geometry'] = mapping(poly)
        # Attributwerte
        elem['properties'] = {'Name': 'Polygon ' + str(cnt),
'Id' : str(cnt)}
        # Erstellen des neuen Datensatzes / Records
        layer.write(elem)

```

Zur Übung kann nun folgendes Beispiel erstellt werden:

Erstelle einen Punktlayer mit n zufällig verteilten Punkten.



## 11. Visualisierung von Gedaten mit Folium

Mit Folium können Geodaten auf einfache Art und Weise in einem interaktiven Map-Container in einem Browser visualisiert werden. Dabei liegt der Fokus auf der einfachen Visualisierung und weniger auf einer komplexen Darstellung. Es können zwar auch thematische Karten erstellt werden, dies aber hauptsächlich auf einigermassen einfache Art lediglich mit Flächendaten. Mit Punktdaten und Liniendaten wird es komplizierter bzw. müssen sog. „Workarounds“ gefunden werden. Nichtsdestotrotz dient Folium einer raschen Visualisierung. Dazu ist es am besten, die darzustellenden Daten in ein GeoJSON Format zu bringen. Dies kann mittels des ogr2ogr Befehls erfolgen.

```
import os
os.system('ogr2ogr -f "GeoJSON" -t_srs "EPSG:4326"
Daten/GEM_SO_WGS84.json Daten/Gemeinden_Solothurn.shp ')
```

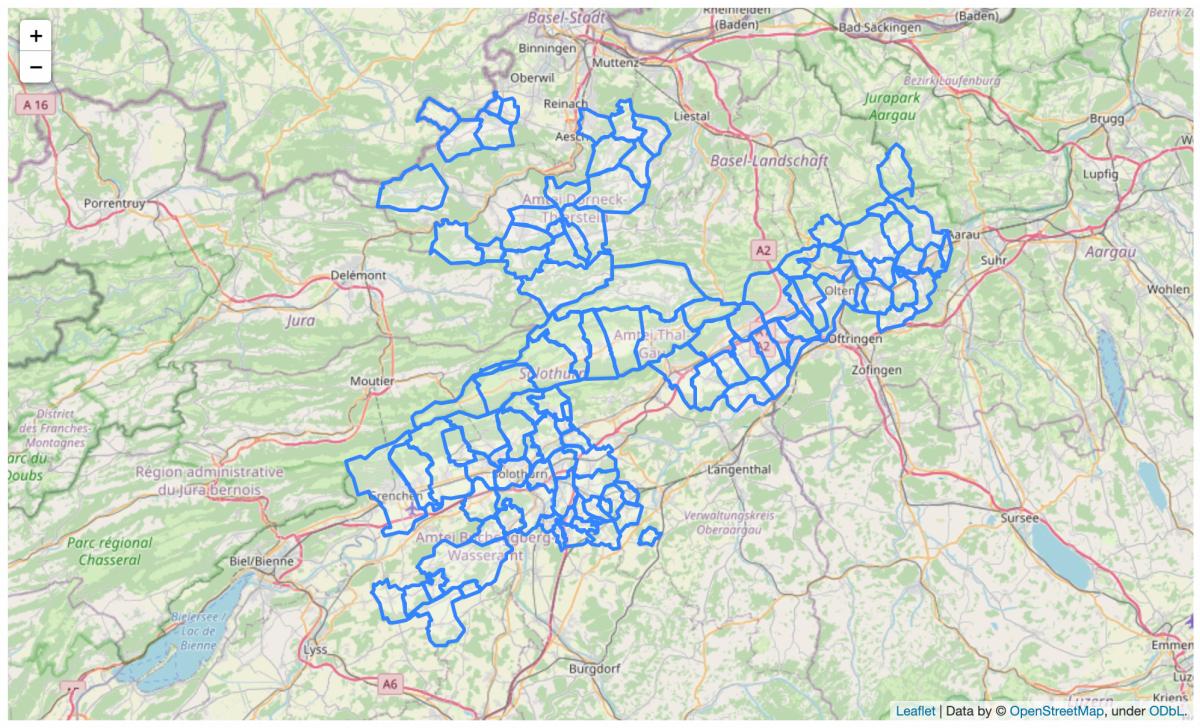
Anschliessend kann mittels folgendem Code ein Folium Objekt erstellt werden:

```
import folium
m = folium.Map(location=[47.27075, 7.70023], zoom_start=10)
```

Danach können die gewünschten Daten in diesen Map-Container integriert werden:

```
# Reading Polygondata - municipalities
rfile = open('../Data/Gemeinden_SolothurnWGS84.json', 'r',
encoding='utf-8').read()
jsonData = json.loads(rfile)                      # Reading as dictionary
style_function = {
    'fillColor': 'white',
}
folium.GeoJson(jsonData, name='json_data',
               style_function=lambda x: style_function      #
style_function has to be a function which calls dictionary
).add_to(m)           # Overlay on map
```

Das Ergebnis sieht danach wie folgt aus:



**Abbildung 16 Darstellung der Gemeinden des Kantons Solothurn in einem Webbrowser mittels Folium**

Analog zu den Flächendaten kann auch ein Punktobjekt als Marker hinzugefügt werden:

```
folium.Marker(  
    location=[47.40875, 8.50778],  
    popup="ETH",  
    icon=folium.Icon(color="red", icon="info-sign"),  
).add_to(m)
```

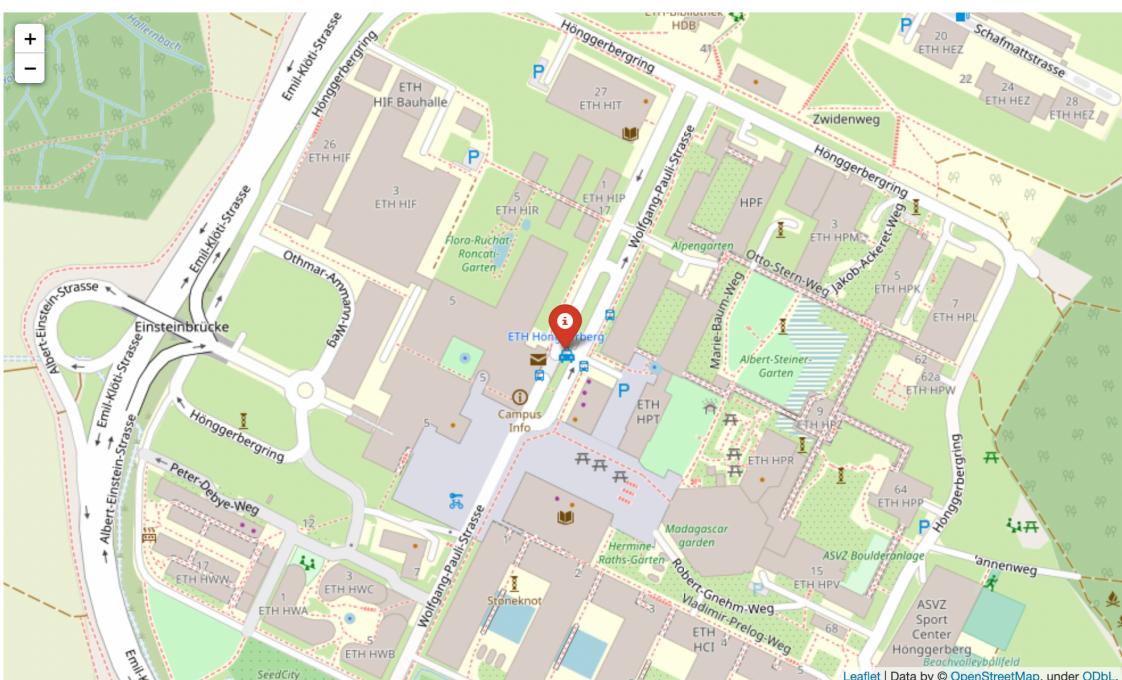


Abbildung 17 Darstellung eines Punktes in Form eines Markers in einem Webbrowser mittels Folium

Die Kartenkomposition kann als HTML Datei mit folgendem Befehl gespeichert werden:

```
m.save('..../Data/myMap.html')
```

Weitere Informationen zum Einsatz von Folium:

<https://python-visualization.github.io/folium/>

<https://www.analyticsvidhya.com/blog/2020/06/guide-geospatial-analysis-folium-python/>

## 12. Abschlussübung

Openrouteservice stellt kostenlos einen Webdienst zur Verfügung für die Berechnung diverser Routing-Aufgaben. Unter anderem können Einzugsgebiete oder Fahrzeitonen (Isochronen) berechnet werden (<https://openrouteservice.org>). Die angegebene Webseite enthält auch eine Dokumentation zur API mit Quellcode-Beispielen für einen erfolgreichen Aufruf bzw. Nutzung der Dienste.

Aufgabe: Erstellen Sie ein Python-Programm, welches

- den Isochronendienst nutzt
- für einen oder mehrere Standorte Fahrzeitonen berechnet
- die (individuelle) Angabe von Fahrzeiten erlaubt
- die erhaltene Geometrie in ein anderes Format konvertiert (bspw. Shape oder GeoPackage)
- es zulässt, dass für mehrere Standorte und mehrere Fahrzeiten die Ergebnisse nach Fahrzeiten getrennt in einzelne Dateien gespeichert werden
- ...