Geodatenanalyse mit Python (SS2017)

Johannes Schwer (johannes.schwer@geo.uni-augsburg.de)

Hausaufgabe 3

Stand 22.06.2017

Abgabe im Digicampus bis 02.07.2017.

Die erfolgreiche Bearbeitung dieser unbenoteten Hausaufgabe ist als Studienleistung Zugangsvoraussetzung für die Prüfungsleistung am Ende des Semesters.

Die Aufgabenstellungen sind bewusst (1) offen formuliert und (2) in manchen Belangen anspruchsvoll.

Das bedeutet, dass Sie nicht für alle Aspekte eine fertige Lösung in den Kursunterlagen finden werden. Grundsätzlich haben Sie aber im bisherigen Verlauf der Veranstaltung wesentliche Konzepte und "Bausteine" kennengelernt, die Sie sinnvoll kombinieren und durch Selbststudium unter Zuhilfenahme zur Verfügung stehender Hilfsmittel (Dokumentationen, Google/Stackoverflow o.ä., Kommilitonen, Sprechstunde)... ergänzen sollen.

Eine "korrekte" oder "perfekte" Lösung gibt es in diesem Fall nicht. Insofern bedeutet eine "erfolgreiche" Bearbeitung der Hausaufgabe, dass Sie versuchen zu einer Lösung kommen und dabei v.a. Ihren Gedanken-/Entscheidungsprozess verdeutlichen. Bewertungskriterien sind somit einerseits die formal sorgfältige Bearbeitung (z.B. ordentliche und aussagekräftige Docstrings) als auch andererseits der Nachweis von Transfer- und Problemlösungskompetenz...

Auch wenn etwa Aspekte der Implementierung nicht funktionieren sollten, widerspricht dies also nicht grundsätzlich einer "erfolgreichen" Bearbeitung. Dokumentieren Sie Ihre Ideen sowie (ggf. erfolglose) Lösungsstrategien.

http://66.media.tumblr.com/19cdeeaead3727d459f7282d39742857/tumblr_nj5u5r7QKx1s9cjaio1_400.jpg

Flugverkehr

In dieser Hausaufgabe arbeiten Sie primär mit den openflights-Daten, um die Objektorientierte Programmierung sowie den Umgang mit verschiedenen Geodaten-Formaten mit Python einzuüben.

Geben Sie im Digicampus ein Zip-Archiv ab, das Ihr Python-Modul/Ihre Python-Module sowie die beiden Ergebnis-Shapefiles aus **c)** enthält.

Es gibt keine "perfekte Lösung", stattdessen führen viele Wege nach Rom...

Gutes Gelingen!

a) OO-Modellierung

Setzen Sie sich mit den openflights-Daten

- airports.dat und airports-extended.dat
- routes.dat
- airlines.dat

und den dazugehörigen Metadaten auseinander. In den kommenden Sitzungen wollen wir damit nicht zuletzt Strukturen im globalen Luftverkehr mithilfe von Netzwerkanalysen untersuchen. Grundsätzlich wären auch Simulationen des Flugverkehrs oder Echtzeitinformationssysteme wie FlightRadar o.ä. denkbare Anwendungen.

Leiten Sie in diesem Kontext ein objektorientiertes Modell zur Repräsentation der vorgefundenen Datenstrukturen ab und implementieren Sie dieses mithilfe der bereits thematisierten Strukturen (Klassen, Klassen- und Instanzvariablen, Methoden, ...):

- Welche Entitäten gibt es und wie stehen diese zueinander in Beziehung?
- Welche Attribute und "Fähigkeiten" sind zu erfassen bzw. zu implementieren?
- Erkennen Sie "implizite" Entitäten, die in den Daten nicht explizit enthalten sind, die sinnvollerweise berücksichtigt werden sollten?
- Gibt es mehr als nur eine Herangehensweise? Diskutieren Sie etwaige Abwägungen und Entscheidung beim Design Ihres Modelles.

Begründen Sie Ihre Modellierung, indem Sie Ihren Code ausführlich kommentieren. Benutzen Sie dafür *Docstrings* und orientieren Sie sich bei der Gestaltung dieser an den Konventionen aus PEP 257 https://www.python.org/dev/peps/pep-0257/.

Demonstrieren Sie die Funktionalität Ihres Modelles durch Beispielaufrufe innerhalb der main-Funktion Ihres Moduls. Sie können die folgende Vorlage benutzen:

In [ ]:

```
# -*- coding: utf-8 -*-
"""
Modul-Docstring.

@author: rz-kennung/Name
"""

class XY(object):
    """
    Klassen-Docstring.
    """

    def __init__(self):
        """
        Methoden-Docstring.
        """
        pass

# ...

if __name__ == "__main__":
    """
    Main-Funktion

    Jedes Python-Modul bekommt beim Aufruf einen "__name__" definiert:
    Dieser ist "__main__" falls das Modul als "standalone"-Programm ausgefuehrt
 wird.
    Wird das Modul (z.B. xyz.py) von anderer Stelle importiert, dann wird "__nam
e__" auf "xyz" gesetzt.
    Daraus folgt, dass die Main-Funktion nur ausgefuehrt wird,
    falls das Modul als eigenstaendiges Programm gestartet wird,
    nicht aber falls es importiert wird.
    """

    # Beispielaufruf
    foo = XY()
```

b) Import/Export

1.: Das Modell mit Daten füttern

Das Modell soll nun mit Daten gefüttert und zum Leben erweckt werden. Entwickeln Sie Möglichkeiten zur Instanziierung Ihres Modells auf Basis unterschiedlicher Datenformate:

- CSV-Dateien (Hinweis zu Airports: Benutzen Sie die Datei "airports-extended.dat" und filtern Sie Airports ueber das Attribut "Type" heraus)
- HTML-Tabelle (http://terrapreta.geo.uni-augsburg.de/airports.html; nur Airports, keine Routes/Airlines)
- optional: JSON/GeoJSON, Shapefile

2.: Daten exportieren

Entwickeln Sie ebenso Möglichkeiten zum sinnvollen Export der Datenstrukturen:

- Shapefile
- GeoJSON (siehe Exkurs zu GeoJSON am Ende des Dokumentes)

Manche Entitäten haben womöglich keinen direkten Raumbezug, dann sind andere Formate gefragt. Schaffen Sie daher eine Möglichkeit zur Speicherung der erzeugten Python-Objekte auf der Festplatte, damit diese noch nach Programmende behalten und bei Bedarf wieder geladen werden können (Objekt-Persistenz/-Serialisierung). Recherchieren Sie dafür die built-in Module:

- pickle
- json

Demonstrieren Sie die Funktionalitäten Ihres Modelles durch Beispielaufrufe innerhalb der main-Funktion Ihres Moduls.

c) Analyse

Bearbeiten Sie die folgenden Fragestellungen:

- Wieviele Airport-Features gibt es pro Land? Benutzen Sie Verschneidungs-Funktionalitäten von OGR sowie das selbst erzeugte Airports-Shapefile und das Natural Earth Countries-Shapefile. Speichern Sie die Anzahl pro Land in einem neuen Attributfeld (Integer) in einer Kopie des Countries-Shapefiles.
- Werten Sie aus, wieviele Routen pro Airport ein- und ausgehen. Schreiben Sie die Anzahl ausgehender als auch die Anzahl eingehender Routen als neue Attribute in das Airport-Shapefile.

Exkurs GeoJSON

GeoJSON

– GeoJSON = JavaScript Object Notation
  - Open-standard format derived from JavaScript
  - Human-readable and easy to parse by computers
  - Increasinlgy important format in web development
  - Largely replacing markup languages like XML in Web GIS context
  - Objects are described by **“key”: value** pairs:

```
{
  "id": 1,
  "name": "Foo",
  "price": 123,
  "tags": [
    "Bar",
    "Eek"
  ],
  "stock": {
    "warehouse": 300,
    "retail": 20
  }
}
```

https://en.wikipedia.org/wiki/JSON

GeoJSON

– GeoJSON builds up on JSON and extends it for
  - representing Simple Features geometry types:
    Point, LineString, Polygon, MultiPoint, MultiLineString,
    and MultiPolygon
  - along with their non-spatial attributes:
    Geometric objects with additional properties are Feature objects:

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [125.6, 10.1]
  },
  "properties": {
    "name": "Dinagat Islands"
  }
}
```

Source: http://www.geojson.org/

GeoJSON Geometry Primitives


```
{ "type": "Point",
    "coordinates": [30, 10]
}
```

```
{ "type": "LineString",
    "coordinates": [ [30, 10], [10, 30], [40, 40] ]
}
```

```
{ "type": "Polygon",
    "coordinates": [ [[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]] ]
}
```

```
{ "type": "Polygon",
    "coordinates": [
        [[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],
        [[20, 30], [35, 35], [30, 20], [20, 30]] ]
}
```

https://en.wikipedia.org/wiki/GeoJSON

GeoJSON Multipart Geometries

```
{ "type": "MultiPoint",
    "coordinates": [ [10, 40], [40, 30], [20, 20], [30, 10] ]
}
```

```
{ "type": "MultiLineString",
    "coordinates": [
        [[10, 10], [20, 20], [10, 40]],
        [[40, 40], [30, 30], [40, 20], [30, 10]]
    ]
}
```

```
{ "type": "MultiPolygon",
    "coordinates": [
        [
            [[30, 20], [45, 40], [10, 40], [30, 20]]
        ],
        [
            [[15, 5], [40, 10], [10, 20], [5, 10], [15, 5]]
        ]
    ]
}
```

```
{ "type": "MultiPolygon",
    "coordinates": [
        [
            [[40, 40], [20, 45], [45, 30], [40, 40]]
        ],
        [
            [[20, 35], [10, 30], [10, 10], [30, 5], [45, 20], [20, 35]],
            [[30, 20], [20, 15], [20, 25], [30, 20]]
        ]
    ]
}
```

GeoJSON

– Sets of features are contained by FeatureCollection objects

```
{ "type": "FeatureCollection",
  "features": [
    { "type": "Feature",
      "geometry": {"type": "Point", "coordinates": [102.0, 0.5]},
      "properties": {"prop0": "value0"}
      },
    { "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
          ]
        },
      "properties": {
        "prop0": "value0",
        "prop1": 0.0
        }
      },
    { "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
            [100.0, 1.0], [100.0, 0.0] ]
          ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": {"this": "that"}
        }
      }
    ]
  }
```

Source: http://www.geojson.org/

Tipp: Sie können die Korrektheit Ihrer GeoJSON-Features auf dieser Seite unkompliziert testen: http://geojson.io