Data Analyst Nanodegree

Project 2

Data Wrangling with MongoDB

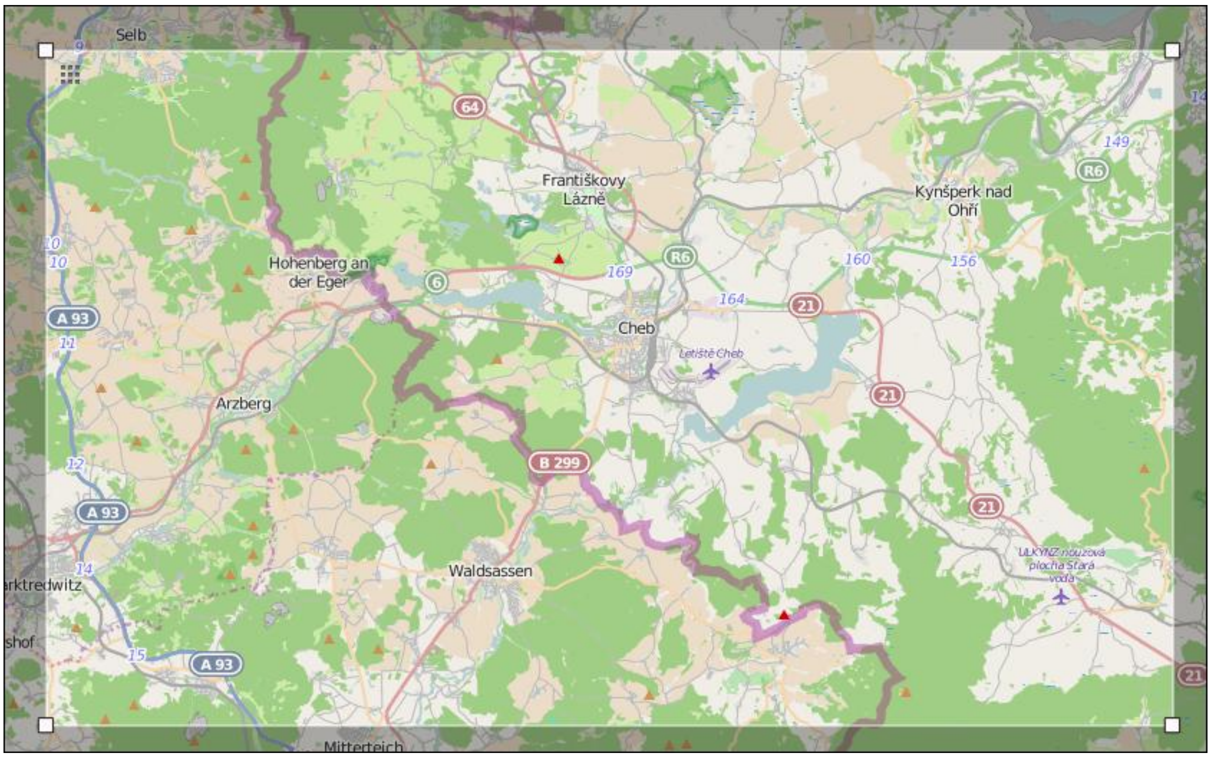
OpenStreetMap: District Cheb

Michal Mašika

1. Introduction

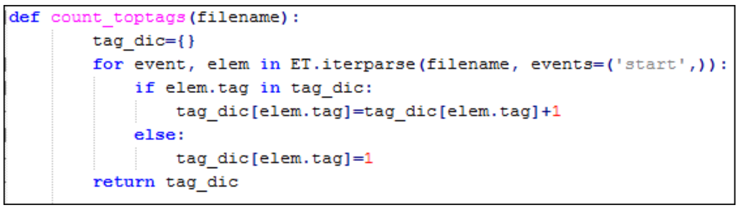
Data wrangling is a very important part of data science as there is a huge amount of “raw” unstructured data in the world. Before analyzing it must be transformed into a convenient format. It is a process consisting of gathering, extracting, cleaning and storing our data. This project is the second project of Udacity data analyst nanodegree program. I demonstrate here some skills which I gathered in the class “Data Wrangling with MongoDB”.

I was born and grew up in Cheb, which is a small Czech town on the border to Germany. Therefore, it was pretty clear what area I am going to investigate for this project. To this end I went to http://openstreetmap.org and search for Cheb. After that I chose the boundaries, i.e. maximum and minimum for longitude and latitude, such that the final osm.file is sufficiently large. The boundaries are as follows: min latitude="49.9572", min longitude="12.0918", max latitude="50.1644" and max longitude="12.6308". The size of the file is almost 114 MB. The picture below shows the examining area. We can see there that the area contains several towns from both sides of the border.



2. Data Cleaning (Problems encountered in the map)

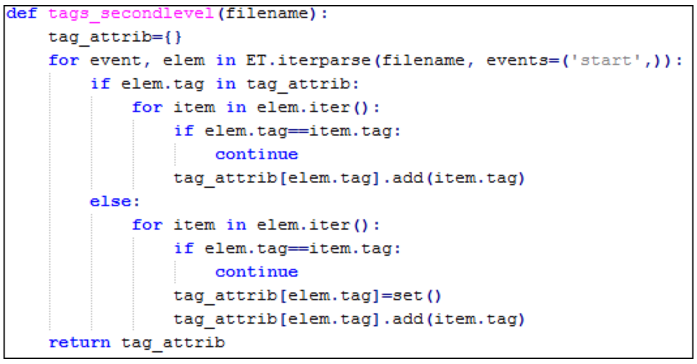
Before I process the data into MongoDB it is very useful to audit it. In general, I was pretty impressed how “clean” (although incomplete) this data is. In very first step I was wondering what tags are there. To this end, I used count_tags.py. The result can be seen below:



```python
def count_toptags(filename):
    tag_dic={}
    for event, elem in ET.iterparse(filename, events=('start',)):
        if elem.tag in tag_dic:
            tag_dic[elem.tag]=tag_dic[elem.tag]+1
        else:
            tag_dic[elem.tag]=1
    return tag_dic
```

```
{'bounds': 1,
 'member': 77593,
 'meta': 1,
 'nd': 1362209,
 'node': 1178040,
 'note': 1,
 'osm': 1,
 'relation': 2827,
 'tag': 705745,
 'way': 118262}
```

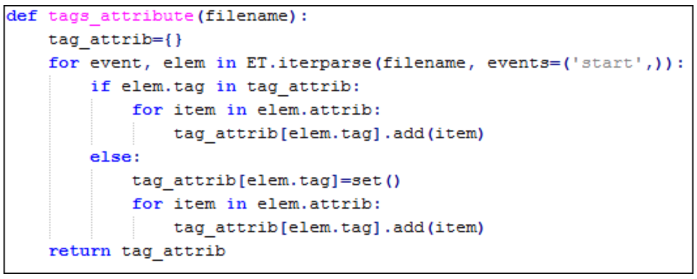
As expected, the most used tags are node, way, tag and nd. Moreover, some of these tags also contain nested tags:



```python
def tags_secondlevel(filename):
    tag_attrib={}
    for event, elem in ET.iterparse(filename, events=('start',)):
        if elem.tag in tag_attrib:
            for item in elem.iter():
                if elem.tag==item.tag:
                    continue
                tag_attrib[elem.tag].add(item.tag)
        else:
            for item in elem.iter():
                if elem.tag==item.tag:
                    continue
                tag_attrib[elem.tag]=set()
                tag_attrib[elem.tag].add(item.tag)
    return tag_attrib
```

```
{'node': set(['tag']),
 'osm': set(['node']),
 'relation': set(['member', 'tag']),
 'way': set(['nd', 'tag'])}
```

For the later cleaning as well as for the analysis it can be useful to look at the attributes of each of the above tags. This is done in tags_attrib.py:



```
def tags_attribute(filename):
    tag_attrib={}
    for event, elem in ET.iterparse(filename, events=('start',)):
        if elem.tag in tag_attrib:
            for item in elem.attrib:
                tag_attrib[elem.tag].add(item)
        else:
            tag_attrib[elem.tag]=set()
            for item in elem.attrib:
                tag_attrib[elem.tag].add(item)
    return tag_attrib
```

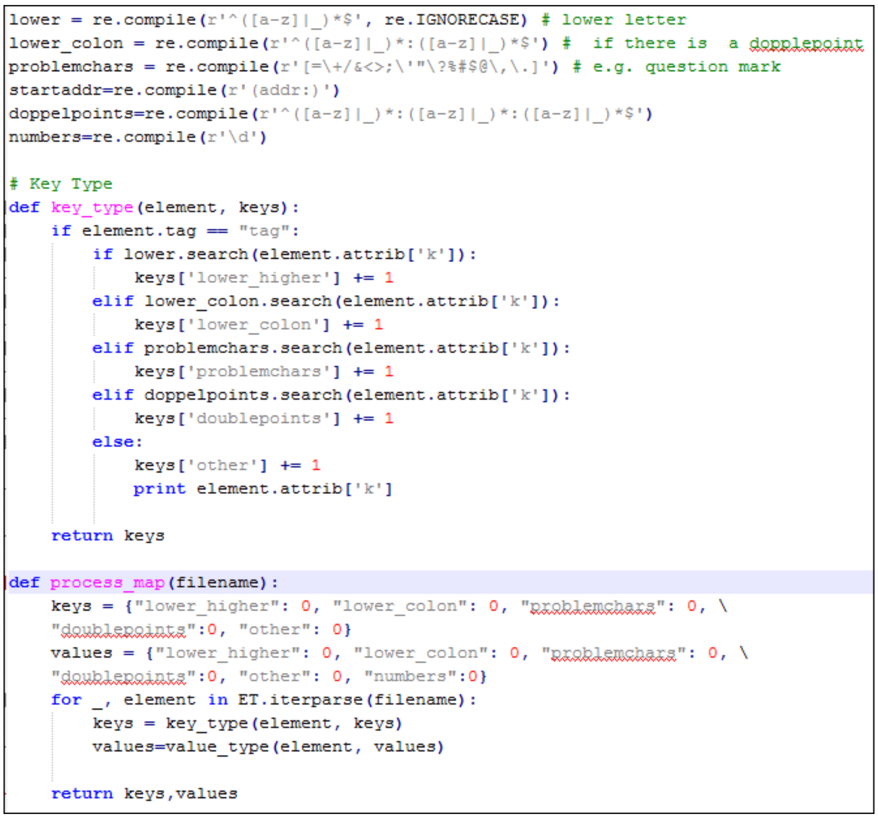
```
{'bounds': set(['maxlat', 'maxlon', 'minlat', 'minlon']),
 'member': set(['ref', 'role', 'type']),
 'meta': set(['osm_base']),
 'nd': set(['ref']),
 'node': set(['changeset', 'id', 'lat', 'lon', 'timestamp', 'uid', 'user', 'version']),
 'note': set([]),
 'osm': set(['generator', 'version']),
 'relation': set(['changeset', 'id', 'timestamp', 'uid', 'user', 'version']),
 'tag': set(['k', 'v']),
 'way': set(['changeset', 'id', 'timestamp', 'uid', 'user', 'version'])}
```

To the most used attributes belong those attributes which can be found by nodes, ways and relations.

```
{'changeset': 553910,
 'generator': 1,
 'id': 553910,
 'k': 278036,
 'lat': 504247,
 'lon': 504247,
 'maxlat': 1,
 'maxlon': 1,
 'minlat': 1,
 'minlon': 1,
 'osm_base': 1,
 'ref': 640934,
 'role': 59393,
 'timestamp': 553910,
 'type': 59393,
 'uid': 553910,
 'user': 553910,
 'v': 278036,
 'version': 553911}
```

The most interesting attributes are keys (‘k’) and values (‘v’) as these provide us with additional useful information about the map. Thus, in the next step I examine the keys and values in more detail. Hereby I am using keys_values.py. In very first step I examine whether there are some problematic characters in keys and/or values. To this end I make use of regular expressions defined in the same way as in the lesson 6:[1]

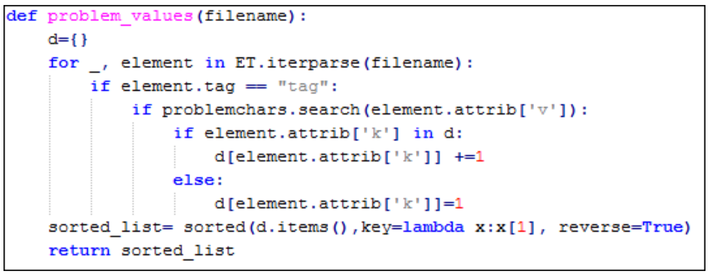
[1] The procedure for examining the types of values is very similar to the key_type. So, I decided not to include it here.



```
lower = re.compile(r'^([a-z]|_)*$', re.IGNORECASE) # lower letter
lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$') #  if there is  a dopplepoint
problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\.]') # e.g. question mark
startaddr=re.compile(r'(addr:)')
doppelpoints=re.compile(r'^([a-z]|_)*:([a-z]|_)*:([a-z]|_)*$')
numbers=re.compile(r'\d')

# Key Type
def key_type(element, keys):
    if element.tag == "tag":
        if lower.search(element.attrib['k']):
            keys['lower_higher'] += 1
        elif lower_colon.search(element.attrib['k']):
            keys['lower_colon'] += 1
        elif problemchars.search(element.attrib['k']):
            keys['problemchars'] += 1
        elif doppelpoints.search(element.attrib['k']):
            keys['doublepoints'] += 1
        else:
            keys['other'] += 1
            print element.attrib['k']

    return keys

def process_map(filename):
    keys = {"lower_higher": 0, "lower_colon": 0, "problemchars": 0, \
    "doublepoints":0, "other": 0}
    values = {"lower_higher": 0, "lower_colon": 0, "problemchars": 0, \
    "doublepoints":0, "other": 0, "numbers":0}
    for _, element in ET.iterparse(filename):
        keys = key_type(element, keys)
        values=value_type(element, values)

    return keys,values
```

Keys:	**Values:**
{'doublepoints': 12235, 'lower_colon': 139807, 'lower_higher': 125284, 'other': 710, 'problemchars': 0}	{'doublepoints': 33, 'lower_colon': 16202, 'lower_higher': 125810, 'numbers': 89941, 'other': 25124, 'problemchars': 20926}

We can see that while there are no problematic characters among keys there are many values with problematic characters. In the next step I look at the keys with the problematic characters in their values. Below you can find the keys with the highest occurrence of problematic characters in their values (behind the key we can see the number of caseswith problematic values). Some of them I will examine and clean later in this chapter:



```
def problem_values(filename):
    d={}
    for _, element in ET.iterparse(filename):
        if element.tag == "tag":
            if problemchars.search(element.attrib['v']):
                if element.attrib['k'] in d:
                    d[element.attrib['k']] +=1
                else:
                    d[element.attrib['k']]=1
    sorted_list= sorted(d.items(),key=lambda x:x[1], reverse=True)
    return sorted_list
```

[('addr:housenumber', 9169), ('is_in', 9067), ('source', 492), ('name', 272), ('addr:street', 250), ('website', 143), ('addr:city', 130), ('TMC:cid_58:tabcd_1:LCLversion', 96), ('ref', 93), ('phone', 77)]

There are 832 keys in total. There are a lot of keys (e.g. recycling:metal, seats, disused:shop, except …) which are used only ones. The following keys belong to the 10 most used in the dataset:

```
[('source', 27216), ('building', 24731), ('addr:postcode', 18836), ('addr:city', 18257), ('addr:housenumber', 17659), ('addr:country', 16381), ('highway', 11515), ('addr:street', 10909), ('addr:place', 10720), ('addr:conscriptionnumber', 9250)]
```

There are a lot of possibilities how to audit this data. Here I present some of them

i. Country (i.e. addr:country)

In the first step I can audit validity of some of our data. To this end I look at the values of country as from the discussion above it is clear that there can be only two values: CZ or DE. In the dataset, all the values are correctly specified and therefore no updating is necessary.

ii. House, Conscription and Street number

Moreover, I can audit validity by examining cross field constraints. In Czech Republic the house number is equal to conscription number/street number (i.e. hn=333/84, cn=333, sn=84). Thus, I can examine whether this relationship is always fulfilled. By using procedure addr_numbers in keys_values.py we can see that there are no problems with this relationship if all of these keys exist. Nevertheless, there are situations, where one or another number is missing. For example, there is information about house number and conscription number, but the street number is missing. We can make use of the relationship above and add the missing key into the data (see update_strnumber(d) in keys_values.py).[2]

iii. City (i.e. addr:city)

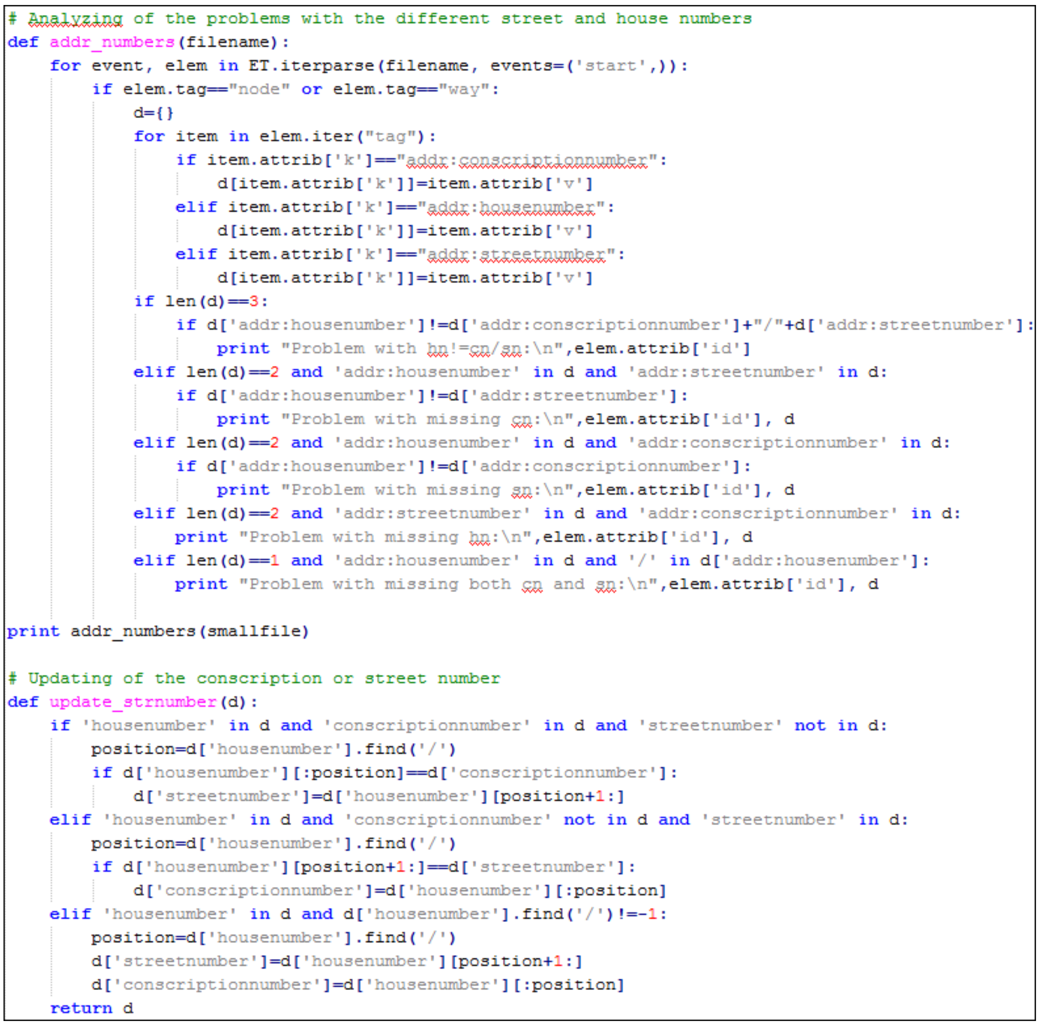
By examining the cities we observe that there are two names for town "Hohenberg an der Eger" (1. Hohenberg an der Eger and 2. Hohenberg a. d. Eger). I changed the second name to the first one, such that our data is consistent. This was the only problem with the cities in our dataset.

iv. Post Code (i.e. addr:postcode)

There is only one inconsistency problem with post code. The post code "351 32" is written with a white space inside, while the others contain no white space inside. Thus, I remove the white space from the problematic post code.[3]

---

[2] The procedure update_strnumber(d) is then used in very last step before we import the data into MongoDB.
[3] Note that the problematic post code is in the tag "relation" which is not considered in this project. The new code is oldcode.replace(" ","").



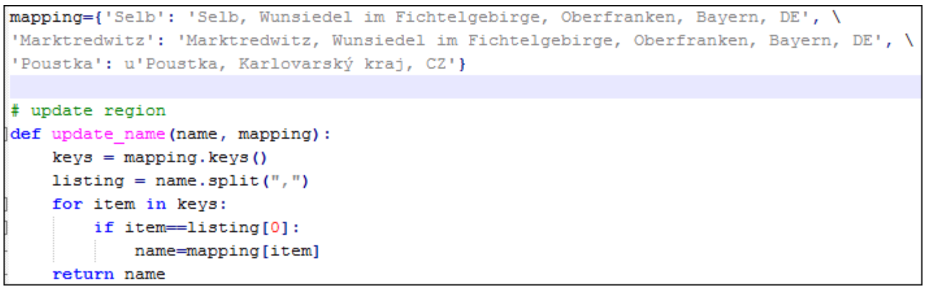
```
# Analyzing of the problems with the different street and house numbers
def addr_numbers(filename):
    for event, elem in ET.iterparse(filename, events=('start',)):
        if elem.tag=="node" or elem.tag=="way":
            d={}
            for item in elem.iter("tag"):
                if item.attrib['k']=="addr:conscriptionnumber":
                    d[item.attrib['k']]=item.attrib['v']
                elif item.attrib['k']=="addr:housenumber":
                    d[item.attrib['k']]=item.attrib['v']
                elif item.attrib['k']=="addr:streetnumber":
                    d[item.attrib['k']]=item.attrib['v']
            if len(d)==3:
                if d['addr:housenumber']!=d['addr:conscriptionnumber']+"/"+d['addr:streetnumber']:
                    print "Problem with hn!=cn/sn:\n",elem.attrib['id']
            elif len(d)==2 and 'addr:housenumber' in d and 'addr:streetnumber' in d:
                if d['addr:housenumber']!=d['addr:streetnumber']:
                    print "Problem with missing cn:\n",elem.attrib['id'], d
            elif len(d)==2 and 'addr:housenumber' in d and 'addr:conscriptionnumber' in d:
                if d['addr:housenumber']!=d['addr:conscriptionnumber']:
                    print "Problem with missing sn:\n",elem.attrib['id'], d
            elif len(d)==2 and 'addr:streetnumber' in d and 'addr:conscriptionnumber' in d:
                print "Problem with missing hn:\n",elem.attrib['id'], d
            elif len(d)==1 and 'addr:housenumber' in d and '/' in d['addr:housenumber']:
                print "Problem with missing both cn and sn:\n",elem.attrib['id'], d

print addr_numbers(smallfile)

# Updating of the conscription or street number
def update_strnumber(d):
    if 'housenumber' in d and 'conscriptionnumber' in d and 'streetnumber' not in d:
        position=d['housenumber'].find('/')
        if d['housenumber'][:position]==d['conscriptionnumber']:
            d['streetnumber']=d['housenumber'][position+1:]
    elif 'housenumber' in d and 'conscriptionnumber' not in d and 'streetnumber' in d:
        position=d['housenumber'].find('/')
        if d['housenumber'][position+1:]==d['streetnumber']:
            d['conscriptionnumber']=d['housenumber'][:position]
    elif 'housenumber' in d and d['housenumber'].find('/')!=-1:
        position=d['housenumber'].find('/')
        d['streetnumber']=d['housenumber'][position+1:]
        d['conscriptionnumber']=d['housenumber'][:position]
    return d
```

v. Region (i.e. is_in)

The key "is_in" contains useful information about the region which the object is in. Unfortunately, this information is not always complete. To complete this information I proceed in the following steps. First, I examined how the structure of the value looks like. Second, I filtered out the problematic values. Third, I corrected the problematic values. The structure of "is_in" for objects in Czech Republic is "part of the town, town, region, country".[4] Key "is_in" of German objects contains information about town, county, administrative region, state and country. You can find two examples below. In addition, you can see below also an illustrative example how updating of the region looks like. This is then done right before importing into MongoDB.

Czech	**German**
Hradiště, Cheb, Karlovarský kraj, CZ	Selb, Wunsiedel im Fichtelgebirge, Oberfranken, Bayern, DE

[4] Note that it wasn't always possible to find out, what is the part of the town. Therefore, sometimes "is_in" contains only information about town, region and country.



```
mapping={'Selb': 'Selb, Wunsiedel im Fichtelgebirge, Oberfranken, Bayern, DE', \
'Marktredwitz': 'Marktredwitz, Wunsiedel im Fichtelgebirge, Oberfranken, Bayern, DE', \
'Poustka': u'Poustka, Karlovarský kraj, CZ'}

# update region
def update_name(name, mapping):
    keys = mapping.keys()
    listing = name.split(",")
    for item in keys:
        if item==listing[0]:
            name=mapping[item]
    return name
```

vi. Street (i.e. addr:street)

Streets belong to the most important ingredients of the map. Therefore, it is very important to check whether names of the streets are correct and consistent. Unfortunately, we cannot use the same procedure as in the lesson 6. The reason is that the name of Czech streets consists often only from one word (e.g. Palackého and not Palackého ulice).[5] Nevertheless, I modified a little bit the code from lesson 6, in order to examine whether there are any other problems with street names (e.g. abbreviations). The analysis didn't indicate any such problems. Moreover, we saw above that in 250 cases the street name contains problematic characters. I examined this in more detail, in order to see if any modification is necessary. Below you can find the problematic streets with number of their occurrence in the data. If we go through the streets, we can see that there are no problems with them and therefore no update is necessary.

17. listopadu	35	5. května	28	26. dubna	24	J. K. Tyla	23
K. Havlíčka Borovského	23	nám. Republiky	21	St.-Georg-Straße	19	Jana A. Komenského	13
Dr.-Martin-Luther-Straße	13	Dr. Pohoreckého	12	Malé nám.	9	V. B. Třebízského	6
náměstí J. W. Goetheho	6	náměstí 5. května	5	P.-Mauritius-Straße	2	Dr. Šimka	3
J. E. Purkyně	4	G'steinigt	2	Dr.-Heim-Straße	1	Dr.-M.-Lutherstraße	1

Finally, there is a possibility that some streets from the region are missing in the map. In order to examine this issue, I can make a cross check with additional sources. This procedure is described in the last section where I present some other ideas about the dataset.

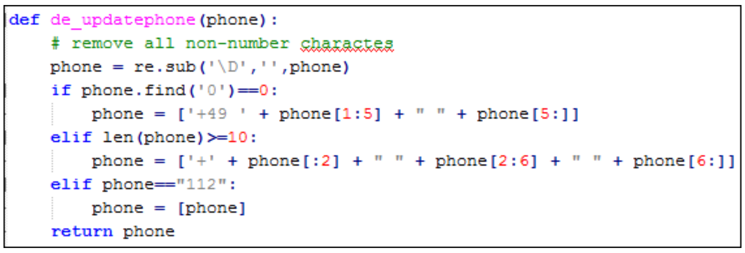
vii. Phone (i.e. phone)

We saw in the previous analysis that there are values with problematic characters by key "phone". This is primarily due to white spaces and characters like "/" or "+". Moreover, we can easily see that the values are not consistent. In order to achieve consistency in my data, I update the values of phone. The updated value is a list of phone numbers. While the Czech phone numbers are stored in the format +420 xxx xxx xxx, the format of German phone

[5] Ulice is the Czech name for street.

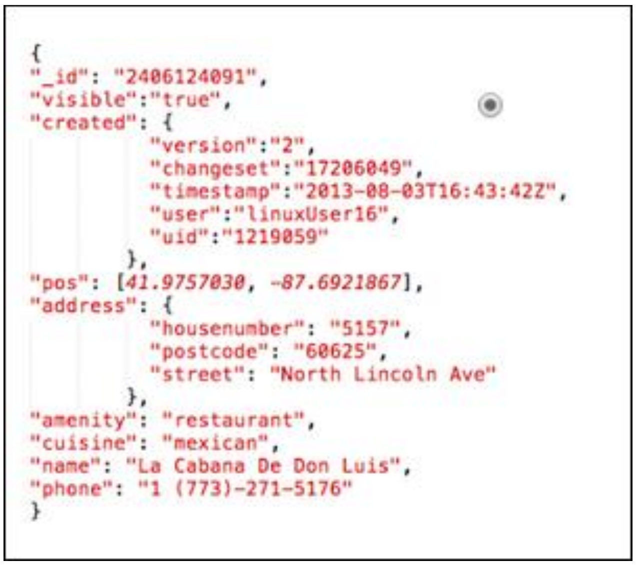
numbers is +49 xxxx xxxx(x). Below you can find some examples of the transformation:[6] Below you can find an example of procedure how German numbers are updated.

	Czech phone numbers		German phone numbers	
Old format	+420 357 071900	+420 354 422 110; +420 775 565 544	09231/9624-0	49 -92 35 - 98 11 0
Updated format	[+420 357 071 900]	[+420 354 422 110, +420 775 565 544]	[+49 9231 96240]	[+49 9235 98110]



```
def de_updatephone(phone):
    # remove all non-number charactes
    phone = re.sub('\D','',phone)
    if phone.find('0')==0:
        phone = ['+49 ' + phone[1:5] + " " + phone[5:]]
    elif len(phone)>=10:
        phone = ['+' + phone[:2] + " " + phone[2:6] + " " + phone[6:]]
    elif phone=="112":
        phone = [phone]
    return phone
```

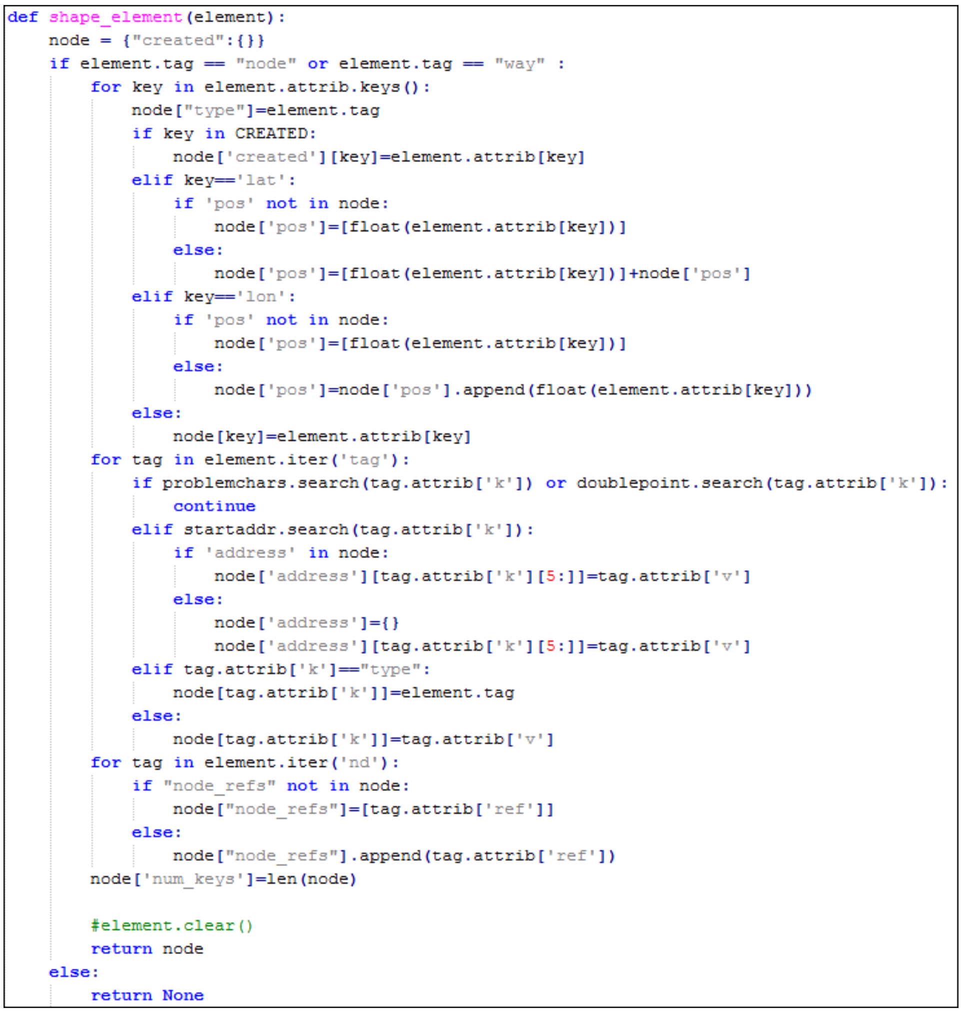
After auditing and updating the data I transform it into a convenient format, such that it can be easily written into the json format. json is important as I can easily import it into MongoDB afterwards. This is done in prepare_db.py. The data format is similar to that one in lesson 6:[7]



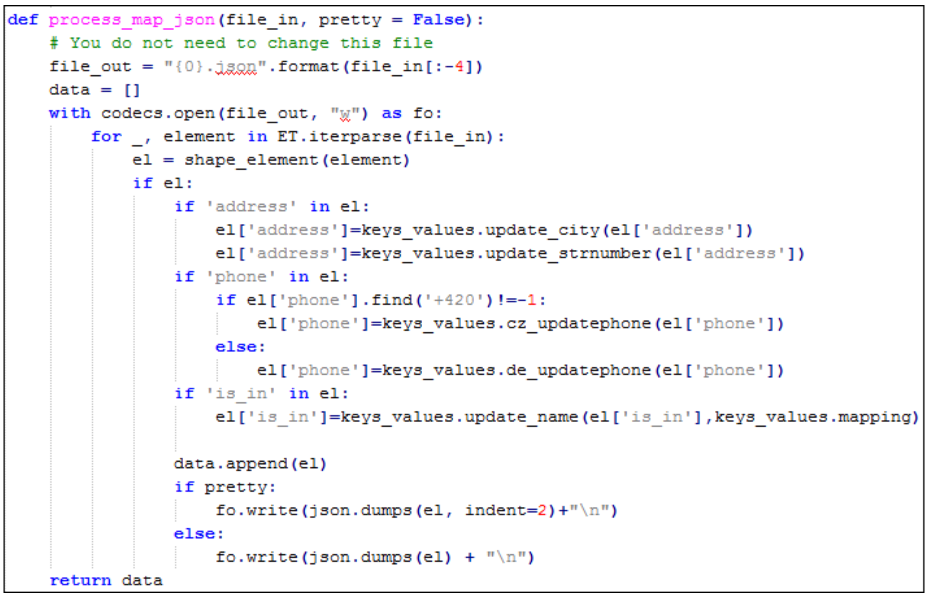
```
{
"_id": "2406124091",
"visible":"true",
"created": {
          "version":"2",
          "changeset":"17206049",
          "timestamp":"2013-08-03T16:43:42Z",
          "user":"linuxUser16",
          "uid":"1219059"
        },
"pos": [41.9757030, -87.6921867],
"address": {
          "housenumber": "5157",
          "postcode": "60625",
          "street": "North Lincoln Ave"
        },
"amenity": "restaurant",
"cuisine": "mexican",
"name": "La Cabana De Don Luis",
"phone": "1 (773)-271-5176"
}
```

[6] Note, that I manually correct one typo in the data. In trail_raiding_station Landhof List there was a wrong phone number (0933/3644). I changed it to 09233/3644.

[7] In 8 cases (documents) there was already key="type". This was updated to way or node respectively.



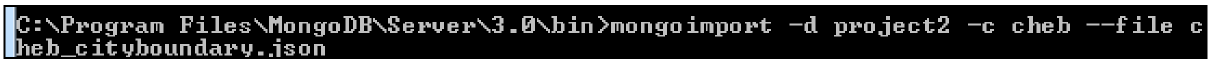
```python
def shape_element(element):
    node = {"created":{}}
    if element.tag == "node" or element.tag == "way" :
        for key in element.attrib.keys():
            node["type"]=element.tag
            if key in CREATED:
                node['created'][key]=element.attrib[key]
            elif key=='lat':
                if 'pos' not in node:
                    node['pos']=[float(element.attrib[key])]
                else:
                    node['pos']=[float(element.attrib[key])]+node['pos']
            elif key=='lon':
                if 'pos' not in node:
                    node['pos']=[float(element.attrib[key])]
                else:
                    node['pos']=node['pos'].append(float(element.attrib[key]))
            else:
                node[key]=element.attrib[key]
        for tag in element.iter('tag'):
            if problemchars.search(tag.attrib['k']) or doublepoint.search(tag.attrib['k']):
                continue
            elif startaddr.search(tag.attrib['k']):
                if 'address' in node:
                    node['address'][tag.attrib['k'][5:]]=tag.attrib['v']
                else:
                    node['address']={}
                    node['address'][tag.attrib['k'][5:]]=tag.attrib['v']
            elif tag.attrib['k']=="type":
                node[tag.attrib['k']]=element.tag
            else:
                node[tag.attrib['k']]=tag.attrib['v']
        for tag in element.iter('nd'):
            if "node_refs" not in node:
                node["node_refs"]=[tag.attrib['ref']]
            else:
                node["node_refs"].append(tag.attrib['ref'])
        node['num_keys']=len(node)

        #element.clear()
        return node
    else:
        return None
```



```python
def process_map_json(file_in, pretty = False):
    # You do not need to change this file
    file_out = "{0}.json".format(file_in[:-4])
    data = []
    with codecs.open(file_out, "w") as fo:
        for _, element in ET.iterparse(file_in):
            el = shape_element(element)
            if el:
                if 'address' in el:
                    el['address']=keys_values.update_city(el['address'])
                    el['address']=keys_values.update_strnumber(el['address'])
                if 'phone' in el:
                    if el['phone'].find('+420')!=-1:
                        el['phone']=keys_values.cz_updatephone(el['phone'])
                    else:
                        el['phone']=keys_values.de_updatephone(el['phone'])
                if 'is_in' in el:
                    el['is_in']=keys_values.update_name(el['is_in'],keys_values.mapping)

                data.append(el)
                if pretty:
                    fo.write(json.dumps(el, indent=2)+"\n")
                else:
                    fo.write(json.dumps(el) + "\n")
    return data
```

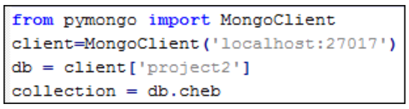
3. Overview of the Data

In the first step I import created json file into MongoDB by using mongoimport in the mongo shell.



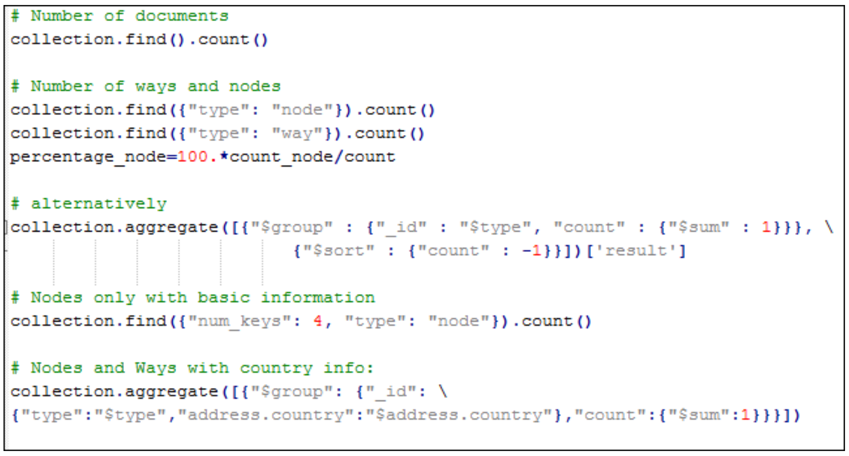
```
C:\Program Files\MongoDB\Server\3.0\bin>mongoimport -d project2 -c cheb --file c
heb_cityboundary.json
```

After the data is imported I can start analyze it. All the analysis is done in data_analyzis.py. While the osm file is 113.891996 MB, the size of the JSON file is 175.490269 MB.



```
from pymongo import MongoClient
client=MongoClient('localhost:27017')
db = client['project2']
collection = db.cheb
```

i. Documents and document types



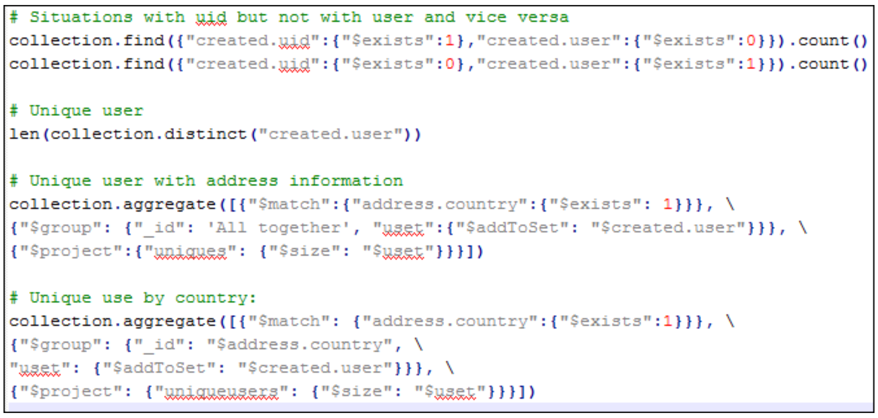
```
# Number of documents
collection.find().count()

# Number of ways and nodes
collection.find({"type": "node"}).count()
collection.find({"type": "way"}).count()
percentage_node=100.*count_node/count

# alternatively
collection.aggregate([{"$group" : {"_id" : "$type", "count" : {"$sum" : 1}}}, \
                           {"$sort" : {"count" : -1}}])['result']

# Nodes only with basic information
collection.find({"num_keys": 4, "type": "node"}).count()

# Nodes and Ways with country info:
collection.aggregate([{"$group": {"_id": \
{"type":"$type","address.country":"$address.country"},"count":{"$sum":1}}}])
```

There are 552 284 documents. While the number of nodes is 504 247 (i.e. 91.3 % of all documents), there are only 48 037 ways. Moreover, the majority of nodes (478 515) contain only basic information (i.e. the following fields: “created”, “id”, “type” and “position”). By 12 872 documents address.country is set to “CZ”. This number is much lower (3 495) for documents where this key is “DE”. You can find an overview in table below.

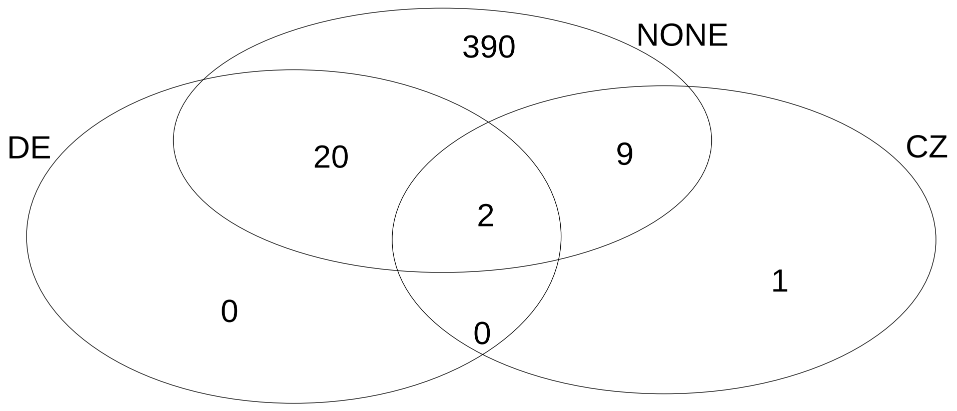
Addr.Country/ Type	**CZ**	**DE**	**None**	**Total**
Node	12 840	1 767	489 640	504 247
Way	32	1 728	46 277	48 037
Total	12 872	3 495	535 917	552 284

ii. Users

There are neither documents with ‘uid’ and without ‘user’ nor documents with ‘user’ and without ‘uid’. Therefore, I can freely choose what key I will use for the queries about user. There are 422 unique users. If I look at the unique users which also entered the explicit information about country, this number amounts to 32.[8] This means that 390 users contributed only basic information to the map. Examining the unique users by country leads to the following numbers: CZ: 12 and DE: 22. It is a little bit surprising that the number for DE is larger than for CZ as we previously saw that the number of documents with CZ as an addr.country is much larger. From these numbers all other numbers can be derived. There is 1 user entered only documents containing “CZ” as an addr.country. On the other hand there are no users entered only information containing “DE” as an addr.country. There are exactly two users supplied documents with all three options (no country, DE and CZ). An illustration can be found below in the picture.

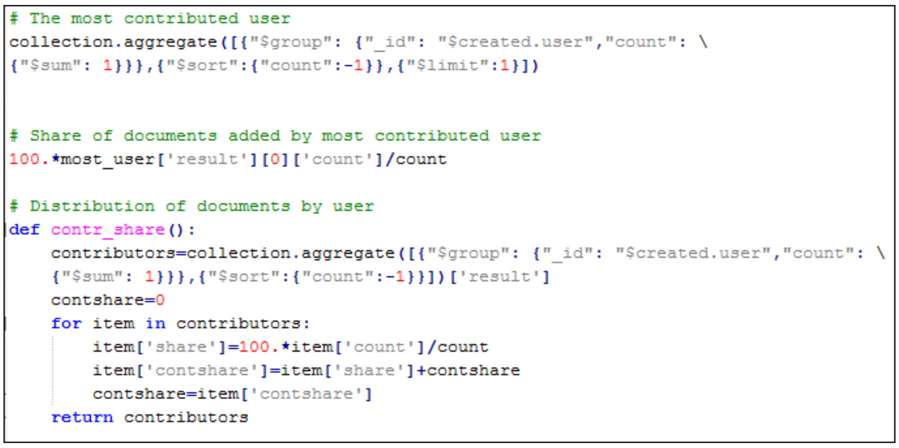


```
# Situations with uid but not with user and vice versa
collection.find({"created.uid":{"$exists":1},"created.user":{"$exists":0}}).count()
collection.find({"created.uid":{"$exists":0},"created.user":{"$exists":1}}).count()

# Unique user
len(collection.distinct("created.user"))

# Unique user with address information
collection.aggregate([{"$match":{"address.country":{"$exists": 1}}}, \
{"$group": {"_id": 'All together', "uset":{"$addToSet": "$created.user"}}}, \
{"$project":{"uniques": {"$size": "$uset"}}}])

# Unique use by country:
collection.aggregate([{"$match": {"address.country":{"$exists":1}}}, \
{"$group": {"_id": "$address.country", \
"uset": {"$addToSet": "$created.user"}}}, \
{"$project": {"uniqueusers": {"$size": "$uset"}}}])
```



[8] The country is of course defined by latitude and longitude.

The most contributed user is geodreieck4711, who has entered 96 450 documents (i.e. 17.46% of all documents). Moreover, the second one is Petr1868 with 62 748 (i.e. 11.36% of all documents). First 10 users provided more than 70 % of all documents.

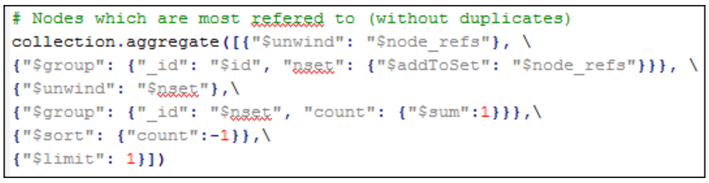


```
# The most contributed user
collection.aggregate([{"$group": {"_id": "$created.user","count": \
{"$sum": 1}}},{"$sort":{"count":-1}},{"$limit":1}])


# Share of documents added by most contributed user
100.*most_user['result'][0]['count']/count

# Distribution of documents by user
def contr_share():
    contributors=collection.aggregate([{"$group": {"_id": "$created.user","count": \
    {"$sum": 1}}},{"$sort":{"count":-1}}])['result']
    contshare=0
    for item in contributors:
        item['share']=100.*item['count']/count
        item['contshare']=item['share']+contshare
        contshare=item['contshare']
    return contributors
```

The same exercise can be done by country. While for the “Czech” documents is Minimalis_import with 8 953 entries (i.e. 69.55%), jacobbraeutigam with 2835 entries (81.12%) is the best contributor for “German” documents.

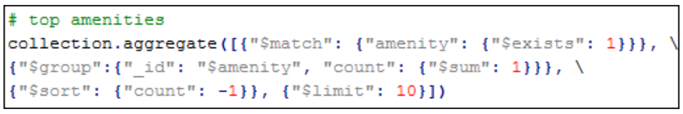
iii. Most referenced node

The ways are ordered list of nodes (at least two). There can be open and closed way. The only difference is that by closed way the first and the last node is the same. Thus, which node is the most referred to? In case of closed way I consider the starting and the end node as one. Node 1366686154 is the most referred to.



```
# Nodes which are most refered to (without duplicates)
collection.aggregate([{"$unwind": "$node_refs"}, \
{"$group": {"_id": "$id", "nset": {"$addToSet": "$node_refs"}}}, \
{"$unwind": "$nset"},\
{"$group": {"_id": "$nset", "count": {"$sum":1}}},\
{"$sort": {"count":-1}},\
{"$limit": 1}])
```

iv. Top 10 amenities



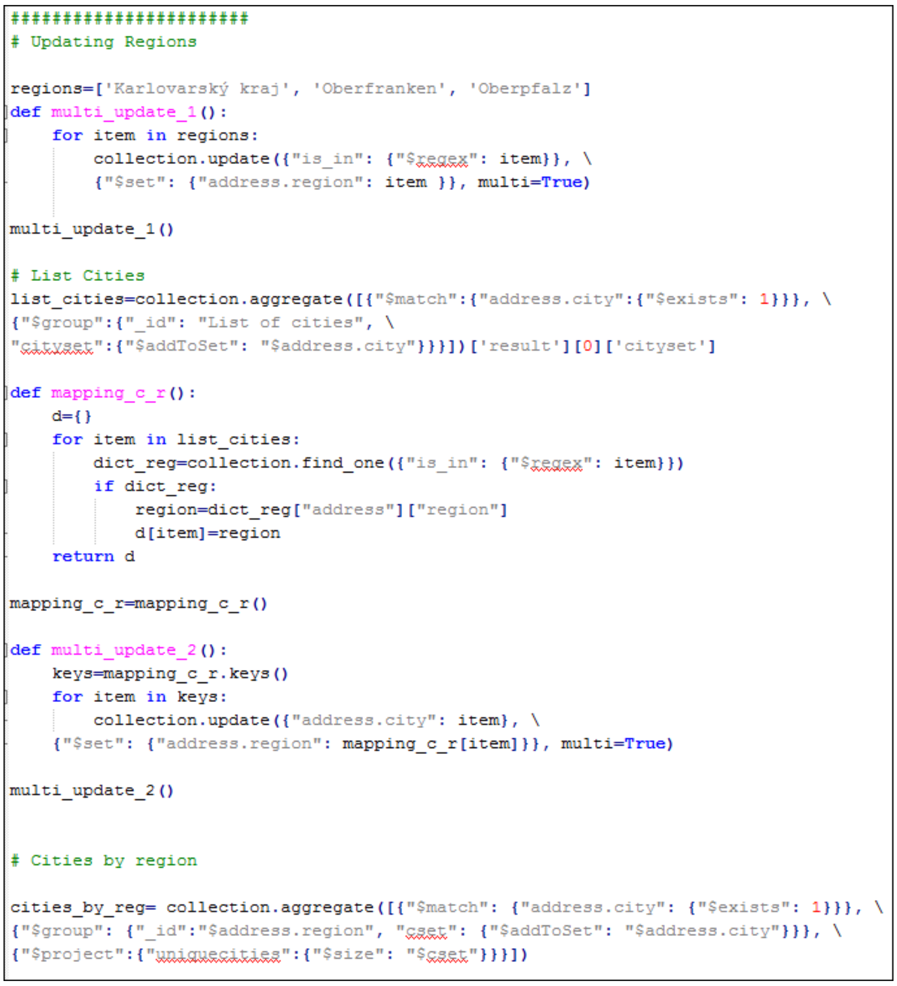
```
# top amenities
collection.aggregate([{"$match": {"amenity": {"$exists": 1}}}, \
{"$group":{"_id": "$amenity", "count": {"$sum": 1}}}, \
{"$sort": {"count": -1}}, {"$limit": 10}])
```

In the following table you find the ten top amenities in my map. The most mentioned amenity is parking. You can find it in 252 documents.

1. parking (252)	2. bench (194)	3. hunting_stand (102)	4. restaurant (87)	5. place_of_worship (66)
6. grit_bin (48)	7. post_box (34)	8. fuel (32)	9. recycling (30)	10. cafe (27)

v. Cities by region

There are three regions in my map: Karlovarský kraj, Oberfranken and Oberpfalz. The question is how many cities we observe in each of these three regions. In order to examine this, in the first step I update data by creating new key “region”. In order to create this field I am using keys “is_in” and “address.city”.



```
#######################
# Updating Regions

regions=['Karlovarský kraj', 'Oberfranken', 'Oberpfalz']
def multi_update_1():
    for item in regions:
        collection.update({"is_in": {"$regex": item}}, \
        {"$set": {"address.region": item }}, multi=True)

multi_update_1()

# List Cities
list_cities=collection.aggregate([{"$match":{"address.city":{"$exists": 1}}}, \
{"$group":{"_id": "List of cities", \
"cityset":{"$addToSet": "$address.city"}}}])['result'][0]['cityset']

def mapping_c_r():
    d={}
    for item in list_cities:
        dict_reg=collection.find_one({"is_in": {"$regex": item}})
        if dict_reg:
            region=dict_reg["address"]["region"]
            d[item]=region
    return d

mapping_c_r=mapping_c_r()

def multi_update_2():
    keys=mapping_c_r.keys()
    for item in keys:
        collection.update({"address.city": item}, \
    {"$set": {"address.region": mapping_c_r[item]}}, multi=True)

multi_update_2()


# Cities by region

cities_by_reg= collection.aggregate([{"$match": {"address.city": {"$exists": 1}}}, \
{"$group": {"_id":"$address.region", "cset": {"$addToSet": "$address.city"}}}, \
{"$project":{"uniquecities":{"$size": "$cset"}}}])
```

In the second step I compute the numbers of cities by region. Note that I was not able to assign to each city a region.[9] There are 12 cities without any region information.

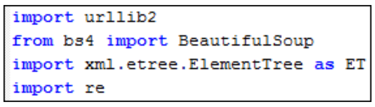
Karlovarský kraj:	30	Oberfranken:	6	Oberpfalz:	3	None	12

[9] Another possibility would be to create key “region” by cleaning the data (before importing into MongoDB).

4. Conclusion (additional ideas)

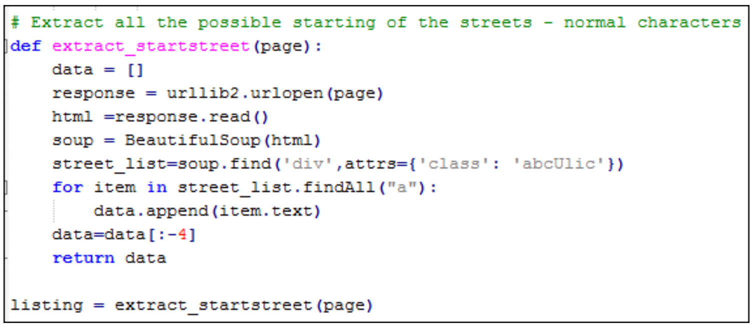
In this project I gathered data about an area around Czech city Cheb from openstreetmap.org. I clean it up and stored in MongoDB. Moreover, I performed some data analysis in order to get a good data overview. The review has shown that the data is incomplete. Nevertheless, I believe that the dataset was well cleaned for the purposes of this exercise.

There are still several opportunities how to clean and validate the data. One possibility would be to cross-check it with other sources (e.g. googlemaps). Here, I decided to examine whether the list of streets in Cheb is "complete". To this end I scrape the list of streets in Cheb from http://www.psc.cz/cz/Karlovarsky-kraj/Cheb/Cheb/



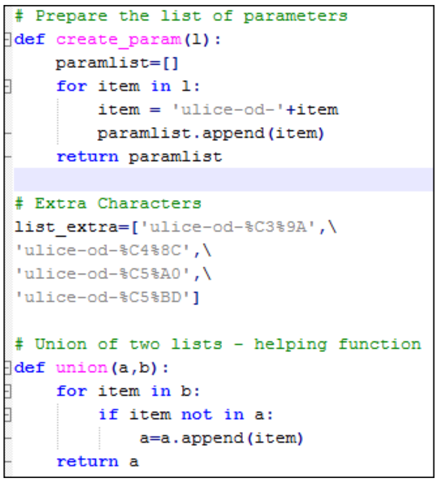
```python
import urllib2
from bs4 import BeautifulSoup
import xml.etree.ElementTree as ET
import re
```

In the first step I need to extract all necessary parameters from the page that I use for the lopping over the page.



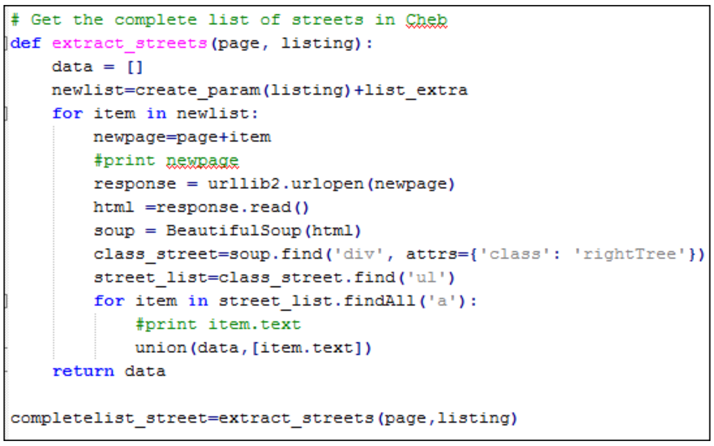
```python
# Extract all the possible starting of the streets - normal characters
def extract_startstreet(page):
    data = []
    response = urllib2.urlopen(page)
    html =response.read()
    soup = BeautifulSoup(html)
    street_list=soup.find('div',attrs={'class': 'abcUlic'})
    for item in street_list.findAll("a"):
        data.append(item.text)
    data=data[:-4]
    return data

listing = extract_startstreet(page)
```

In order to extract streets from the web, I need two help procedures.



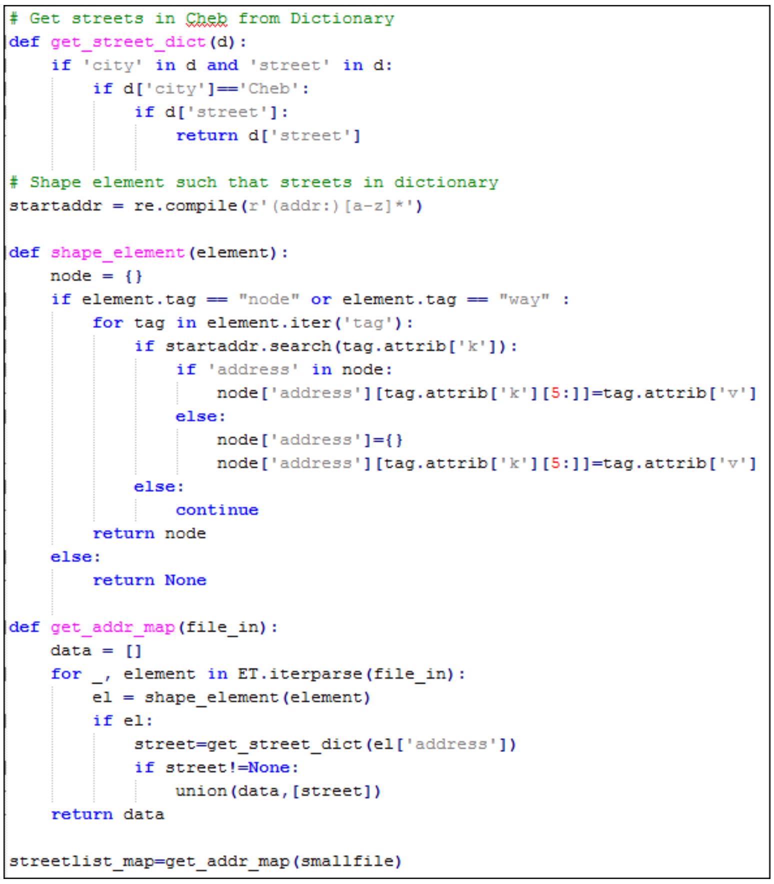
```python
# Prepare the list of parameters
def create_param(l):
    paramlist=[]
    for item in l:
        item = 'ulice-od-'+item
        paramlist.append(item)
    return paramlist

# Extra Characters
list_extra=['ulice-od-%C3%9A',\
'ulice-od-%C4%8C',\
'ulice-od-%C5%A0',\
'ulice-od-%C5%BD']

# Union of two lists - helping function
def union(a,b):
    for item in b:
        if item not in a:
            a=a.append(item)
    return a
```

Afterwards the streets are extracted and put into the list "completelist_street".



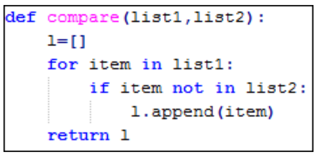
```python
# Get the complete list of streets in Cheb
def extract_streets(page, listing):
    data = []
    newlist=create_param(listing)+list_extra
    for item in newlist:
        newpage=page+item
        #print newpage
        response = urllib2.urlopen(newpage)
        html =response.read()
        soup = BeautifulSoup(html)
        class_street=soup.find('div', attrs={'class': 'rightTree'})
        street_list=class_street.find('ul')
        for item in street_list.findAll('a'):
            #print item.text
            union(data,[item.text])
    return data

completelist_street=extract_streets(page,listing)
```

In the next step I create a list of streets from my map.



```python
# Get streets in Cheb from Dictionary
def get_street_dict(d):
    if 'city' in d and 'street' in d:
        if d['city']=='Cheb':
            if d['street']:
                return d['street']

# Shape element such that streets in dictionary
startaddr = re.compile(r'(addr:)[a-z]*')

def shape_element(element):
    node = {}
    if element.tag == "node" or element.tag == "way" :
        for tag in element.iter('tag'):
            if startaddr.search(tag.attrib['k']):
                if 'address' in node:
                    node['address'][tag.attrib['k'][5:]]=tag.attrib['v']
                else:
                    node['address']={}
                    node['address'][tag.attrib['k'][5:]]=tag.attrib['v']
            else:
                continue
        return node
    else:
        return None

def get_addr_map(file_in):
    data = []
    for _, element in ET.iterparse(file_in):
        el = shape_element(element)
        if el:
            street=get_street_dict(el['address'])
            if street!=None:
                union(data,[street])
    return data

streetlist_map=get_addr_map(smallfile)
```

Finally, I compare the two lists, in order to check if some streets are missing. There are no missing streets in my map.



```python
def compare(list1,list2):
    l=[]
    for item in list1:
        if item not in list2:
            l.append(item)
    return l
```

5. References

Openstreetmap: http://www.openstreetmpa.org

MongoDB documentation: http://docs.mongodb.org/manual/

Updating Regions:

http://de.wikipedia.org/wiki/Thierstein

http://de.wikipedia.org/wiki/Selb

http://de.wikipedia.org/wiki/Thiersheim

http://de.wikipedia.org/wiki/H%C3%B6chst%C3%A4dt_im_Fichtelgebirge

http://de.wikipedia.org/wiki/Hohenberg_an_der_Eger

http://de.wikipedia.org/wiki/Neualbenreuth

http://de.wikipedia.org/wiki/Marktredwitz

Usage for problems with coding:

http://stackoverflow.com/questions/16772071/sort-dict-by-value-python

http://effbot.org/zone/element-iterparse.htm#incremental-parsing

http://stackoverflow.com/questions/7697710/python-running-out-of-memory-parsing-xml-using-celementtree-iterparse

Inspiration from other projects:

https://github.com/allanbreyes/udacity-data-science/blob/master/p2/

https://github.com/orenov/udacity-nanodegree-data-analyst