Using OpenStreetMap data with Python

Andrii V. Mishkovskyi

June 22, 2011

Who is this dude anyway?

- I love Python
- I love OpenStreetMap
- I do map rendering at CloudMade using Python
- CloudMade uses OpenStreetMap data extensively

Objectives

- Understand OpenStreetMap data structure
- How to parse it
- Get a feel of how basic GIS services work

OpenStreetMap

- Founded in 2004 as a response to Ordnance Survey pricing scheme
- >400k registered users
- >16k active mappers
- Supported by Microsoft, MapQuest (AOL), Yahoo!
- Crowd-sourcing at its best

Why OSM?

- Fairly easy
- Good quality
- Growing community
- Absolutely free

Storage type

- XML (.osm)
- Protocol buffers (.pbf, in beta status)
- Other formats through 3rd parties (Esri shapefile, Garmin GPX, etc.)

The data

- Each object has geometry, tags and changeset information
- Tags are simply a list of key/value pairs
- Geometry definition differs for different types
- Changeset is not interesting when simply using the data (as opposed to editing)

Data types

Node	Geometric point or point of interest
Way	Collection of points
Relation	Collections of objects of any type

Nodes

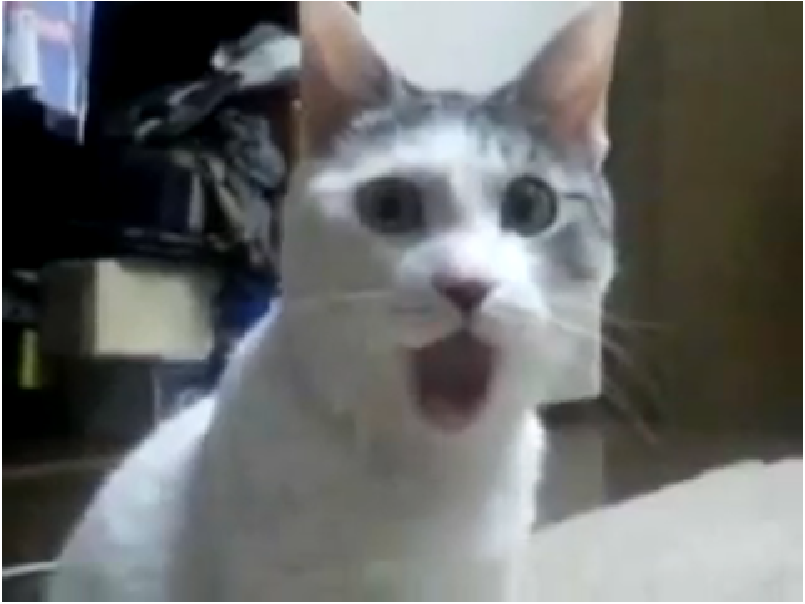

```
<node id="592637238" lat="47.1675211" lon="9.5089882"
      version="2" changeset="6628391"
      user="phinret" uid="135921"
      timestamp="2010-12-11T19:20:16Z">
  <tag k="amenity" v="bar" />
  <tag k="name" v="Black Pearl" />
</node>
```

Ways

```
<way id="4781367" version="1" changeset="102260"
     uid="8710" user="murmel"
     timestamp="2007-06-19T06:25:57Z">
  <nd ref="30604007"/>
  <nd ref="30604015"/>
  <nd ref="30604017"/>
  <nd ref="30604019"/>
  <nd ref="30604020"/>
  <tag k="created_by" v="JOSM" />
  <tag k="highway" v="residential" />
  <tag k="name" v="In den ÃĎusseren" />
</way>
```

Relations


```
<relation id="16239" version="699" changeset="8440520"
          uid="122406" user="hanskuster"
          timestamp="2011-06-14T18:53:49Z">
  <member type="way" ref="75393767" role="outer"/>
  <member type="way" ref="75393837" role="outer"/>
  <member type="way" ref="75393795" role="outer"/>
  ...
  <member type="way" ref="75393788" role="outer"/>
  <tag k="admin_level" v="2" />
  <tag k="boundary" v="administrative" />
  <tag k="currency" v="EUR" />
  <tag k="is_in" v="Europe" />
  <tag k="ISO3166-1" v="AT" />
  <tag k="name" v="ÃŰsterreich" />
  ...
  <tag k="wikipedia:de" v="ÃŰsterreich" />
  <tag k="wikipedia:en" v="Austria" />
</relation>
```



Major points when parsing OSM

- Expect faulty data
- Parse iteratively
- Cache extensively
- Order of elements is not guaranteed
- But it’s generally: nodes, ways, relations
- Ids are unique to datatype, not to the whole data set

Parsing data

- Using SAX
- Doing simple reprojection
- Create geometries using Shapely

Parsing data

Projection

```
import pyproj

projection = pyproj.Proj(
    '+proj=merc +a=6378137 +b=6378137'
    '+lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0'
    '+k=1.0 +units=m +nadgrids=@null +wktext'
    '+no_defs')
```

Parsing data

Nodes

```
from shapely.geometry import Point

class Node(object):

    def __init__(self, id, lonlat, tags):
        self.id = id
        self.geometry = Point(projection(*lonlat))
        self.tags = tags
```

Parsing data

Nodes

```
class SimpleHandler(sax.handler.ContentHandler):

    def __init__(self):
        sax.handler.ContentHandler.__init__(self)
        self.id = None
        self.geometry = None
        self.nodes = {}

    def startElement(self, name, attrs):
        if name == 'node':
            self.id = attrs['id']
            self.tags = {}
            self.geometry = map(
                float, (attrs['lon'], attrs['lat']))
        elif name == 'tag':
            self.tags[attrs['k']] = attrs['v']
```

Parsing data

Nodes

```
def endElement(self, name):
    if name == 'node':
        self.nodes[self.id] = Node(self.id,
                                    self.geometry,
                                    self.tags)
        self.id = None
        self.geometry = None
        self.tags = None
```

Parsing data

Ways

```
from shapely.geometry import LineString

nodes = {...} # dict of nodes, keyed by their ids

class Way(object):

    def __init__(self, id, refs, tags):
        self.id = id
        self.geometry = LineString(
            [(nodes[ref].x, nodes[ref].y)
             for ref in refs])
        self.tags = tags
```

Parsing data

Ways

```
class SimpleHandler(sax.handler.ContentHandler):

    def __init__(self):
        ...
        self.ways = {}

    def startElement(self, name, attrs):
        if name == 'way':
            self.id = attrs['id']
            self.tags = {}
            self.geometry = []
        elif name == 'nd':
            self.geometry.append(attrs['ref'])
```

Parsing data

Ways

```
def reset(self):
    self.id = None
    self.geometry = None
    self.tags = None

def endElement(self, name):
    if name == 'way':
        self.way[self.id] = Way(self.id,
                                self.geometry,
                                self.tags)
        self.reset()
```

Parsing data

Relations

```
from shapely.geometry import MultiPolygon, MultiLineString,

ways = {...} # dict of ways, with ids as keys

class Relation(object):

    def __init__(self, id, members, tags):
        self.id = id
        self.tags = tags
        if tags['type'] == 'multipolygon':
            outer = [ways[member['ref']]
                     for member in members
                     if member['role'] == 'outer']
            inner = [ways[member['ref']]
                     for member in members
                     if member['role'] == 'inner']
            self.geometry = MultiPolygon([(outer, inner)])
```

Parsing data

Relations

The importing code is left as an exercise for the reader

For language zealots

Excuse me for not using namedtuples.

Parsing data: homework

- The idea is simple
- The implementation can use ElementTree if you work with small extracts of data
- Have to stick to SAX when parsing huge extracts or the whole planet data

Existing solutions

- Osmosis
- osm2pgsql
- osm2mongo, osm2shp, etc.

Principles

- Scale
- Projection
- Cartography
- Types of maps

Layers

- Not exactly physical layers
- Layers of graphical representation
- Don't render text in several layers

How to approach rendering

- Split your data in layers
- Make projection configurable
- Provide general way to select data sources
- Think about cartographers

The magic of Mapnik

```
import mapnik

map = mapnik.Map(1000, 1000)
mapnik.load_map(map, "style.xml")
bbox = mapnik.Envelope(mapnik.Coord(-180.0, -90.0),
                       mapnik.Coord(180.0, 90.0))
map.zoom_to_box(bbox)
mapnik.render_to_file(map, 'map.png', 'png')
```

Magic?

- Mapnik's interface is straightforward
- The implementation is not
- Complexity is hidden in XML

Mapnik's XML

```
<Style name="Simple">
  <Rule>
    <PolygonSymbolizer>
      <CssParameter name="fill">#f2eff9
      </CssParameter>
    </PolygonSymbolizer>
    <LineSymbolizer>
      <CssParameter name="stroke">red
      </CssParameter>
      <CssParameter name="stroke-width">0.1
      </CssParameter>
    </LineSymbolizer>
  </Rule>
</Style>
```

Mapnik's XML

```
<Layer name="world" srs="+proj=latlong +datum=WGS84">
  <StyleName>My Style</StyleName>
  <Datasource>
    <Parameter name="type">shape
    </Parameter>
    <Parameter name="file">world_borders
    </Parameter>
  </Datasource>
</Layer>
```

What's that?

- Codename geocoding
- Similar to magnets
- Fast or correct – choose one

Why is it hard?

- Fuzzy search
- Order matters
- But not always
- One place can have many names
- One name can correspond to many places
- People don't care about this at all!

Why is it hard?

I blame Google.

Attempt at implementation

- Put restrictions
- Make the request structured
- Or at least assume order
- Assume valid input from users

Attempt at implementation

```
def geocode(**query):
    boundary = world
    for key in ['country', 'zip', 'city',
                'street', 'housenumber']:
        try:
            value = query[key]
            boundary = find(key, value, boundary)
        except KeyError:
            continue
    return boundary

def find(key, value, boundary):
    for tags, geometry in data:
        if geometry in boundary and\
                tags.get(key) == value:
            return geometry
```

Fixing user input

Soundex/Metaphone/DoubleMetaphone

- Phonetic algorithms
- Works in 90% of the cases
- If your language is English
- Doesn't work well for placenames

Fixing user input

```
from itertools import groupby

def soundex(word):
    table = {'b': 1, 'f': 1, 'p': 1, 'v': 1,
             'c': 2, 'g': 2, 'j': 2, ...}
    yield word[0]
    codes = (table[char]
             for char in word[1:]
             if char in table)
    for code in groupby(codes):
        yield code
```

Fixing user input

Edit distance

- Works for two words
- Most geocoding requests consist of several words
- Scanning database for each pair distance isn't feasible
- Unless you have it cached already
- Check out Peter Norvig's "How to Write Spelling a Corrector" article

Fixing user input

N-grams

- Substrings of n items from the search string
- Easier to index than edit distance
- Gives less false positives than phonetic algorithm
- Trigrams most commonly used

Fixing user input

```
from itertools import izip, islice, tee

def nwise(iterable, count=2):
    iterables = enumerate(tee(iterable, count))
    return izip(*[islice(iterable, start, None)
                  for start, iterables in iterables])

def trigrams(string):
    string = ''.join(['  ', string, ' ']).lower()
    return nwise(string, 3)
```

Making the search free-form

- Normalize input: remove the, a, . . .
- Use existing free-form search solution
- Combine ranks from different sources

Making the search free-form

```
from operator import itemgetter
from collections import defaultdict

def freeform(string):
    ranks = defaultdict(float)
    searchfuncs = [(phonetic, 0.3),
                   (levenshtein, 0.15),
                   (trigrams, 0.55)]
    for searchfunc, coef in searchfuncs:
        for match, rank in searchfunc(string):
            ranks[match] += rank * coef
    return max(ranks.iteritems(), key=itemgetter(1))
```

The problem

When introduced with routing problem, people think Build graph, use Dijsktra, you’re done! (And they are mostly right)

The problem

Not that simple

- Graph is sparse
- Graph has to be updated often
- Dijkstra algorithm is too general
- A* is no better

The problem

- Routing is not only a technical problem
- Different people expect different results for the same input
- Routing through cities is always a bad choice (even if it's projected to be faster)

Building the graph

- Adjacency matrix is not space-efficient
- The graph representation has to very compact
- networkx and igraph are both pretty good for a start

Building the graph

```
from networkx import Graph, shortest_path

...

def build_graph(ways):
    graph = Graph()
    for way, tags in ways:
        for segment in nwise(way.coords):
            weight = length(segment) * coef(tags)
            graph.add_edge(segment[0], segment[1],
                           weight=weight)
    return graph

shortest_path(graph, source, dest)
```

Building the graph

- There is no silver bullet
- No matter how nice these libs are, importing even Europe will require more than 20 GB of RAM
- Splitting data into country graphs is not enough
- Our in-house C++ graph library requires 20GB of mem for the whole world

Other solutions

- PgRouting – easier to start with, couldn't make it fast, harder to configure
- Neo4j – tried 2 years ago, proved to be lacking when presented with huge sparse graphs
- Eat your own dogfood – if doing "serious business", most probably the best solution. Half-wink.

Bored already?

Lighten up, I'm done

Highlights

- Start using OpenStreetMap data – it's easy
- Try building something simple – it's cool
- Try building something cool – it's simple
- Python is one of the best languages [for doing GIS]

Questions?

contact@mishkovskyi.net

Slides: mishkovskyi.net/ep2011