Mopy Tutorial 1

Existing Documentation

Mopy is distributed with an excellent readme file (http://motools.sourceforge.net/mopyreadme.html) covering installation, requirements and basic concepts.

We assume familiarity with that readme file as well as basic Music Ontology concepts and the Python programming language. Due to the intuitive approach of these, you might find it quite easy to read this tutorial even without extensive prior knowledge.

This tutorial...

... gives an example of retrieving data expressed using the chord ontology (http://motools.sourceforge.net/chord_draft_1/chord.html). We are going to use the data stored in an RDF file containing chord descriptions of a particular song, and display them in a human-readable format.

Setup

You will need to have Mopy (comes with motools from http://sourceforge.net/projects/motools/ or from the subversion repository at https://motools.svn.sourceforge.net/svnroot/motools), as well as rdflib (from http://rdflib.net/).

For our example, we create a file called docexample.py, which you can find at http://.../docexample.py together with the complementing helper function file http://.../docexamplefunctions.py. In the following paragraphs we will guide you through the docexample.py file. A sample RDF file is provided at http://.../A_Whiter_Shade_of_Pale.mma.rdf.

Importing the relevant modules

To get going we need some modules provided by the Python and mopy installations, as well as rdflib. <rdflib installation can be quite difficult and should also be documented>

```
# standard modules
import os
from mopy import *

# regular expressions module
import re

# specific mopy submodules
from mopy.timeline import *
from mopy.time import *
from mopy.model import *
from mopy.chord import *
from mopy.mo import *
from mopy.PropertySet import *
from mopy.RDFInterface import importRDFFile

# graph model from the generic RDF module rdflib
from rdflib import ConjunctiveGraph
```

We will also need some helper functions. For now, just download the file http://.../docexamplefunctions.py and save it to the same directory you saved docexample.py to.

```
# our private help function file
from docexamplefunctions import *
```

Once all the imports are dealt with we choose a file to load the song data from.

```
filename = "/Users/MaggieMae/RDF/A_Whiter_Shade_of_Pale.mma.rdf"
```

Loading data

We need a structure for the internal representation of the song and the associated data within Python. That structure is called a conjunctive graph, provided by the rdflib module. We assign an instance of such a graph to g_song:

```
g_song = ConjunctiveGraph()
```

We load the data stored in the song's RDF file into the conjunctive graph:

```
print 'loading the current song'
g_song.load(filename, format="xml")
```

We can now import the conjunctive graph as a RDF graph structure s.

```
print 'constructing RDF graph'
s = importRDFGraph(g_song, False)
```

Additional information from the semantic web

In s, all the information obtained from the RDF file (chords' and chord events' URIs) is stored. However, we can grab more detailed information on the chords from the URIs.

```
# grab chord information.
# s.ChordIdx contains the URI for each chord used in the song (not the chord-
events!)
# we obtain the chord information from the semantic web (knowing the URI)
for c in s.ChordIdx.values():
      print "Loading chord "+str(c.URI)+" ... "
      g_song.load(str(c.URI))

# update the RDF graph
s = importRDFGraph(g_song, False)
```

Retrieve chord events and sort them

As chord events are not sorted automatically, we need to extract their start times and then sort them.

```
# make dictionary of starttimes vs chordevents using the chord event sequence
(s.ChordEventIdx)
time_ce_dict = dict()
pattern_ps = re.compile("P(\d*\.?\d*)S") # format of time as stored in RDF
string
```

If you don't understand this, you might have to learn about regular expressions first, sorry.

Now we loop over the ChordEventIndex and make a look-up table (dictionary) which contains the start time as well as the original event. This table can then be sorted by the dictionary key, i.e. the start time.

```
for ce in s.ChordEventIdx.values():
      starttime_prop = list(ce.time)[0].beginsAtDuration
      starttime = float(pattern_ps.search(list(starttime_prop)[0]).group(1))
      time_ce_dict[starttime] = ce

# sort the dictionary by key (start time of chord event)
```

```
sortedkeys =  time_ce_dict.keys()
sortedkeys.sort()
```

Output the chord event sequence as played in the song

Now that we have a sorted dictionary of chord events, we can loop over it and output any chord associated with the chord events in a human-readable format. That's what we need the helper functions in the file docexamplefunctions.py for: get_notename(), get_intervalsemitone() etc.

```
chord_number = 0
for time_ce in sortedkeys:
      chord_number = chord_number + 1
      current_ce = time_ce_dict[time_ce]
      current_c = list(current_ce.chord)[0]
      print "\n("+str(chord_number)+") at "+str(time_ce)+"s, " #+current_c.URI
```

Get the root note.

```
      if len(current_c.root) > 0:
            current_root = list(current_c.root)[0]
            current_rootname = get_notename(current_root)
            if not current_rootname == None:
                  print "root note: "+ current_rootname
```

Get the bass note (actually this is the relative bass expressed as a musical interval above the root note of the chord).

```
      if len(current_c.bass) > 0:
            current_bass = list(current_c.bass)[0]
            print "bass degree (rel. to root note): " + str(current_bass)
```

Similarly, we need to get an interval list of notes present in the chord, relative to the root note of the chord. The musical intervals can be presented in any order. We would like them to be ordered (by how far away from the root note they are), so we sort them by their “semitone value”.

Notes that are explicitly omitted (like in C7(omit3) in Jazz notation, or C:7(*3) in Chris Harte's syntax which we adapted in the Chord Ontology), have to be subtracted from the list:

```
      interval_list = dict()
      current_intervals = list(current_c.interval)
      for interv in current_intervals:
            interval_list[get_intervalsemitone(interv)] =\
            get_intervalname(interv)

      current_without_intervals = list(current_c.without_interval)
      for interv in current_without_intervals:
            if interval_list.has_key(get_intervalsemitone(interv)):
                  interval_list.pop(get_intervalsemitone(interv))
      keys = interval_list.keys()
      keys.sort()
      values = list()
      for key in keys:
            values.append(interval_list[key])
      interval_tuple = tuple(values)
      print "intervals in the chord: " + str(interval_tuple)
```

Metric duration is an additional information which is not necessarily present in the RDF file. This is duration of a chord measured in beats.

```
      # get metric duration
      durationInt = list(current_ce.time)[0].durationInt
      if not durationInt == None:
            print "duration in beats: "+ str(list(durationInt)[0])
```

Output

After some status information about which chord is currently being loaded and some warnings, which you can choose to ignore, the following output will be obtained (here: abridged):

```
(1) at 0.0s,
root note: C
intervals in the chord: ('1', '3', '5')
duration in beats: 4

(2) at 3.0s,
root note: A
intervals in the chord: ('1', 'b3', '5')
duration in beats: 4

(3) at 6.0s,
root note: F
intervals in the chord: ('1', '3', '5')
duration in beats: 4

....
```