If you come across a list in the data then you just need to call myprint on every element of the list:

**def myprint(d):**

**if isinstance(d,dict): #check if it's a dict before using .iteritems()**

**for k, v in d.iteritems():**

**if isinstance(v, (list,dict)): #check for either list or dict**

**myprint(v)**

**else:**

**print "Key :{0}, Value: {1}".format(k, v)**

**elif isinstance(d,list): #allow for list input too**

**for item in d:**

**myprint(item)**

then you will get an output something like:

...

Key :@name, Value: Employee

Key :@isMandotory, Value: True

Key :#text, Value: Jake Roberts

Key :@name, Value: Section

Key :@isOpen, Value: True

Key :@isMandotory, Value: False

Key :#text, Value: 5

...

Although I'm not sure how useful this is since you have a lot of duplicate keys like @name, I'd like to offer a function I created a while ago to traverse nested json data of nested dicts and lists:

**def traverse(obj, prev\_path = "obj", path\_repr = "{}[{!r}]".format):**

**if isinstance(obj,dict):**

**it = obj.items()**

**elif isinstance(obj,list):**

**it = enumerate(obj)**

**else:**

**yield prev\_path,obj**

**return**

**for k,v in it:**

**for data in traverse(v, path\_repr(prev\_path,k), path\_repr):**

**yield data**

Then you can traverse the data with:

for path,value in traverse(doc):

print("{} = {}".format(path,value))

with the default values for prev\_path and path\_repr it gives output like this:

obj[u'session'][u'@id'] = 2934

obj[u'session'][u'@name'] = Valves

obj[u'session'][u'@docVersion'] = 5.0.1

obj[u'session'][u'docInfo'][u'field'][0][u'@name'] = Employee

obj[u'session'][u'docInfo'][u'field'][0][u'@isMandotory'] = True

obj[u'session'][u'docInfo'][u'field'][0]['#text'] = Jake Roberts

obj[u'session'][u'docInfo'][u'field'][1][u'@name'] = Section

obj[u'session'][u'docInfo'][u'field'][1][u'@isOpen'] = True

obj[u'session'][u'docInfo'][u'field'][1][u'@isMandotory'] = False

obj[u'session'][u'docInfo'][u'field'][1]['#text'] = 5

obj[u'session'][u'docInfo'][u'field'][2][u'@name'] = Location

obj[u'session'][u'docInfo'][u'field'][2][u'@isOpen'] = True

obj[u'session'][u'docInfo'][u'field'][2][u'@isMandotory'] = False

obj[u'session'][u'docInfo'][u'field'][2]['#text'] = Munchen

although you can write a function for path\_repr to take the value of prev\_path (determined by recursively calling path\_repr) and the new key, for example a function to take a tuple and add another element on the end means we can get a (tuple of indices : elem) format which is perfect to pass to the dict constructor

**def \_tuple\_concat(tup, idx):**

**return (\*tup, idx)**

**def flatten\_data(obj):**

**"""converts nested dict and list structure into a flat dictionary with tuple keys**

**corresponding to the sequence of indices to reach particular element"""**

**return dict(traverse(obj, (), \_tuple\_concat))**

**new\_data = flatten\_data(obj)**

**import pprint**

**pprint.pprint(new\_data)**

which gives you the data in this dictionary format:

{('session', '@docVersion'): '5.0.1',

('session', '@id'): 2934,

('session', '@name'): 'Valves',

('session', 'docInfo', 'field', 0, '#text'): 'Jake Roberts',

('session', 'docInfo', 'field', 0, '@isMandotory'): True,

('session', 'docInfo', 'field', 0, '@name'): 'Employee',

('session', 'docInfo', 'field', 1, '#text'): 5,

('session', 'docInfo', 'field', 1, '@isMandotory'): False,

('session', 'docInfo', 'field', 1, '@isOpen'): True,

('session', 'docInfo', 'field', 1, '@name'): 'Section',

('session', 'docInfo', 'field', 2, '#text'): 'Munchen',

('session', 'docInfo', 'field', 2, '@isMandotory'): False,

('session', 'docInfo', 'field', 2, '@isOpen'): True,

('session', 'docInfo', 'field', 2, '@name'): 'Location'}

I found this particularly useful when dealing with my json data but I'm not really sure what you want to do with your xml.