lesson_6_answers

February 22, 2016

1 Lesson 6: Answers to questions

1.0.1 1. mapparser.py:

```
In [ ]: #!/usr/bin/env python
        # -*- coding: utf-8 -*-
        """
        Your task is to use the iterative parsing to process the map file and
        find out not only what tags are there, but also how many, to get the
        feeling on how much of which data you can expect to have in the map.
        Fill out the count_tags function. It should return a dictionary with the
        tag name as the key and number of times this tag can be encountered in
        the map as value.

        Note that your code will be tested with a different data file than the 'example.osm'
        """
        import xml.etree.cElementTree as ET
        import pprint
        from collections import defaultdict

        def count_tags(filename):
                # YOUR CODE HERE
                tagcount = defaultdict(int)
                for line in ET.iterparse(filename):
                    thetag = line[1].tag
                    tagcount[thetag] += 1
                return tagcount



        def test():

            tags = count_tags('example.osm')
            pprint.pprint(tags)
            assert tags == {'bounds': 1,
                             'member': 3,
                             'nd': 4,
                             'node': 20,
                             'osm': 1,
                             'relation': 1,
                             'tag': 7,
                             'way': 1}
```

```
if __name__ == "__main__":
    test()
```

1.0.2 2.

For question 2 my answer was that I prefered the second type of data model.

1.0.3 3. tags.py:

```
In [ ]: #!/usr/bin/env python
        # -*- coding: utf-8 -*-
        import xml.etree.cElementTree as ET
        import pprint
        import re
        """
        Your task is to explore the data a bit more.
        Before you process the data and add it into MongoDB, you should check the "k"
        value for each "<tag>" and see if they can be valid keys in MongoDB, as well as
        see if there are any other potential problems.

        We have provided you with 3 regular expressions to check for certain patterns
        in the tags. As we saw in the quiz earlier, we would like to change the data
        model and expand the "addr:street" type of keys to a dictionary like this:
        {"address": {"street": "Some value"}}
        So, we have to see if we have such tags, and if we have any tags with
        problematic characters.

        Please complete the function 'key_type', such that we have a count of each of
        four tag categories in a dictionary:
          "lower", for tags that contain only lowercase letters and are valid,
          "lower_colon", for otherwise valid tags with a colon in their names,
          "problemchars", for tags with problematic characters, and
          "other", for other tags that do not fall into the other three categories.
        See the 'process_map' and 'test' functions for examples of the expected format.
        """


        lower = re.compile(r'^([a-z]|_)*$')
        lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
        problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')


        def key_type(element, keys):
            if element.tag == "tag":
                k = element.attrib['k']
                if lower.search(k):
                    keys['lower'] +=1
                elif lower_colon.search(k):
                    keys['lower_colon'] +=1
                elif problemchars.search(k):
                    keys['problemchars'] += 1
                else:
                    keys['other'] += 1
```

```
    return keys



def process_map(filename):
    keys = {"lower": 0, "lower_colon": 0, "problemchars": 0, "other": 0}
    for _, element in ET.iterparse(filename):
        keys = key_type(element, keys)

    return keys



def test():
    # You can use another testfile 'map.osm' to look at your solution
    # Note that the assertion below will be incorrect then.
    # Note as well that the test function here is only used in the Test Run;
    # when you submit, your code will be checked against a different dataset.
    keys = process_map('example.osm')
    pprint.pprint(keys)
    assert keys == {'lower': 5, 'lower_colon': 0, 'other': 1, 'problemchars': 1}


if __name__ == "__main__":
    test()
```

1.0.4 4. users.py:

In [ ]:
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import xml.etree.cElementTree as ET
import pprint
import re
"""
Your task is to explore the data a bit more.
The first task is a fun one - find out how many unique users
have contributed to the map in this particular area!

The function process_map should return a set of unique user IDs ("uid")
"""

def get_user(element):

    return element.attrib['uid']


def process_map(filename):
    users = set()
    for _, element in ET.iterparse(filename, events=('start',)):
        if element.tag == "way" or element.tag == "node" or element.tag == "relation":
            users.add(get_user(element))
    return users


def test():

    users = process_map('example.osm')
    pprint.pprint(users)
    assert len(users) == 6



if __name__ == "__main__":
    test()
```

1.0.5 5. audit.py:

In [ ]:
```
"""
Your task in this exercise has two steps:

- audit the OSMFILE and change the variable 'mapping' to reflect the changes needed to fix
    the unexpected street types to the appropriate ones in the expected list.
    You have to add mappings only for the actual problems you find in this OSMFILE,
    not a generalized solution, since that may and will depend on the particular area you are a
- write the update_name function, to actually fix the street name.
    The function takes a string with street name as an argument and should return the fixed nam
    We have provided a simple test so that you see what exactly is expected
"""
import xml.etree.cElementTree as ET
from collections import defaultdict
import re
import pprint

OSMFILE = "example.osm"
street_type_re = re.compile(r'\b\S+\.?$', re.IGNORECASE)


expected = ["Street", "Avenue", "Boulevard", "Drive", "Court", "Place", "Square", "Lane", "Road
            "Trail", "Parkway", "Commons"]

# UPDATE THIS VARIABLE
mapping = { "St": "Street",
            "St.": "Street",
            "Ave":"Avenue",
            "Rd.":"Road"
            }


def audit_street_type(street_types, street_name):
    m = street_type_re.search(street_name)
    if m:
        street_type = m.group()
        if street_type not in expected:
            street_types[street_type].add(street_name)
```

```
def is_street_name(elem):
    return (elem.attrib['k'] == "addr:street")


def audit(osmfile):
    osm_file = open(osmfile, "r")
    street_types = defaultdict(set)
    for event, elem in ET.iterparse(osm_file, events=("start",)):

        if elem.tag == "node" or elem.tag == "way":
            for tag in elem.iter("tag"):
                if is_street_name(tag):
                    audit_street_type(street_types, tag.attrib['v'])
    osm_file.close()
    #pprint.pprint(dict(street_types))
    return street_types


def update_name(name, mapping):

    # YOUR CODE HERE
    m = street_type_re.search(name)
    if m:
        replacethis = m.group()
        name = name.replace(replacethis, mapping[replacethis])


    return name


def test():
    st_types = audit(OSMFILE)
    assert len(st_types) == 3
    pprint.pprint(dict(st_types))

    for st_type, ways in st_types.iteritems():
        for name in ways:
            better_name = update_name(name, mapping)
            print name, "=>", better_name
            if name == "West Lexington St.":
                assert better_name == "West Lexington Street"
            if name == "Baldwin Rd.":
                assert better_name == "Baldwin Road"


if __name__ == '__main__':
    test()
```

1.0.6 6. data.py:

```
In [ ]: #!/usr/bin/env python
        # -*- coding: utf-8 -*-
        import xml.etree.cElementTree as ET
        import pprint
        import re
        import codecs
        import json
        """
        Your task is to wrangle the data and transform the shape of the data
        into the model we mentioned earlier. The output should be a list of dictionaries
        that look like this:

        {
        "id": "2406124091",
        "type: "node",
        "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"
        }

        You have to complete the function 'shape_element'.
        We have provided a function that will parse the map file, and call the function with the elemen
        as an argument. You should return a dictionary, containing the shaped data for that element.
        We have also provided a way to save the data in a file, so that you could use
        mongoimport later on to import the shaped data into MongoDB.

        Note that in this exercise we do not use the 'update street name' procedures
        you worked on in the previous exercise. If you are using this code in your final
        project, you are strongly encouraged to use the code from previous exercise to
        update the street names before you save them to JSON.

        In particular the following things should be done:
        - you should process only 2 types of top level tags: "node" and "way"
        - all attributes of "node" and "way" should be turned into regular key/value pairs, except:
            - attributes in the CREATED array should be added under a key "created"
            - attributes for latitude and longitude should be added to a "pos" array,
              for use in geospacial indexing. Make sure the values inside "pos" array are floats
              and not strings.
        - if the second level tag "k" value contains problematic characters, it should be ignored
        - if the second level tag "k" value starts with "addr:", it should be added to a dictionary "ad
        - if the second level tag "k" value does not start with "addr:", but contains ":", you can
          process it in a way that you feel is best. For example, you might split it into a two-level
          dictionary like with "addr:", or otherwise convert the ":" to create a valid key.
        - if there is a second ":" that separates the type/direction of a street,
          the tag should be ignored, for example:

        <tag k="addr:housenumber" v="5158"/>
        <tag k="addr:street" v="North Lincoln Avenue"/>
        <tag k="addr:street:name" v="Lincoln"/>
        <tag k="addr:street:prefix" v="North"/>
        <tag k="addr:street:type" v="Avenue"/>
        <tag k="amenity" v="pharmacy"/>

          should be turned into:

        {...
        "address": {
            "housenumber": 5158,
            "street": "North Lincoln Avenue"
        }
        "amenity": "pharmacy",
        ...
        }

        - for "way" specifically:

          <nd ref="305896090"/>
          <nd ref="1719825889"/>

        should be turned into
        "node_refs": ["305896090", "1719825889"]
        """


        lower = re.compile(r'^([a-z]|_)*$')
        lower_colon = re.compile(r'^([a-z]|_)*:([a-z]|_)*$')
        problemchars = re.compile(r'[=\+/&<>;\'"\?%#$@\,\. \t\r\n]')

        CREATED = [ "version", "changeset", "timestamp", "user", "uid"]


        def shape_element(element):
            node = {}
            if element.tag == "node" or element.tag == "way" :

                node["id"] = element.attrib["id"]
                node["type"] = element.tag
                if "visible" in element.attrib:
                    node["visible"] = element.attrib["visible"]
                node["created"] = {
                                    "version" : element.attrib["version"],
                                    "changeset": element.attrib["changeset"],
                                    "timestamp": element.attrib["timestamp"],
                                    "user": element.attrib["user"],
                                    "uid": element.attrib["uid"]
                                    }
                if "lat" in element.attrib:
                    node["pos"] = [float(element.attrib["lat"]), float(element.attrib["lon"])]

                address_list = {}
                for tag in element.iter("tag"):

                    m = problemchars.search(tag.attrib['k'])
                    #n = lower_colon.search(tag.attrib['k'])
                    if m:
                        continue
                    elif tag.attrib['k'].startswith("addr:street") and ':' in tag.attrib['k'][10:]:
                        continue
                    elif tag.attrib['k'].startswith("addr"):
                        address_list[tag.attrib['k'][5:]] = tag.attrib['v']
                    elif ':' in tag.attrib['k']:
                        node[tag.attrib['k'].replace(":","_")] = tag.attrib['v']
                    else:
                        node[tag.attrib['k']] = tag.attrib['v']

                if address_list != {}:
                     node["address"] = address_list
                nodes = []
                for anothertag in element.iter("nd"):
                    nodes.append(anothertag.attrib['ref'])
                if nodes != []:
                    node["node_refs"] = nodes



                #pprint.pprint(node)

                return node
            else:
                return None


        def process_map(file_in, pretty = False):
            # You do not need to change this file
            file_out = "{0}.json".format(file_in)
            data = []
            with codecs.open(file_out, "w") as fo:
                for _, element in ET.iterparse(file_in):
                    el = shape_element(element)
                    if el:
                        data.append(el)
                        if pretty:
                            fo.write(json.dumps(el, indent=2)+"\n")
                        else:
                            fo.write(json.dumps(el) + "\n")
            return data

        def test():
            # NOTE: if you are running this code on your computer, with a larger dataset,
            # call the process_map procedure with pretty=False. The pretty=True option adds
            # additional spaces to the output, making it significantly larger.
            data = process_map('example.osm', True)
            #pprint.pprint(data)

            correct_first_elem = {
                "id": "261114295",
                "visible": "true",
                "type": "node",
                "pos": [41.9730791, -87.6866303],
                "created": {
                    "changeset": "11129782",
                    "user": "bbmiller",
                    "version": "7",
                    "uid": "451048",
                    "timestamp": "2012-03-28T18:31:23Z"
                }
            }
            assert data[0] == correct_first_elem
            assert data[-1]["address"] == {
                                            "street": "West Lexington St.",
                                            "housenumber": "1412"
                                              }
            assert data[-1]["node_refs"] == [ "2199822281", "2199822390",  "2199822392", "2199822369",
                                            "2199822370", "2199822284", "2199822281"]

        if __name__ == "__main__":
            test()
```