Data Wrangling with MongoDB
Problem Set Submission

Prerna Singh, 2013149

Problem Set 1 - Data Extraction Fundamentals

Quiz 1 - Using CSV module

#!/usr/bin/env python
"""
Your task is to process the supplied file and use the csv module to extract data from it.
The data comes from NREL (National Renewable Energy Laboratory) website. Each file
contains information from one meteorological station, in particular - about amount of
solar and wind energy for each hour of day.

Note that the first line of the datafile is neither data entry, nor header. It is a line
describing the data source. You should extract the name of the station from it.

The data should be returned as a list of lists (not dictionaries).
You can use the csv modules "reader" method to get data in such format.
Another useful method is next() - to get the next line from the iterator.
You should only change the parse_file function.
"""

SOLUTION

```
import csv
import os

DATADIR = ""
DATAFILE = "745090.csv"


def parse_file(datafile):
    name = ""
    data = []
    count = 0
    with open(datafile,'rb') as f:
        #pass
        myreader = csv.reader(f)
        c = 0
        a1 = myreader.next()[1]
        name = a1
        count=count+1
        b = a1
        myreader.next()
        for i in myreader:
```

```
            count=count+1
            data.append(i)
        #data = list(list(rec) for rec in csv.reader(f, delimiter=','))
    # Do not change the line below
    return (name, data)


def test():
    datafile = os.path.join(DATADIR, DATAFILE)
    name, data = parse_file(datafile)
    #print name
    #print data[0][1]
    #print data
    assert name == "MOUNTAIN VIEW MOFFETT FLD NAS"
    assert data[0][1] == "01:00"
    assert data[2][0] == "01/01/2005"
    assert data[2][5] == "2"


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

Quiz 2 - Excel to CSV

-*- coding: utf-8 -*-
'''
Find the time and value of max load for each of the regions
COAST, EAST, FAR_WEST, NORTH, NORTH_C, SOUTHERN, SOUTH_C, WEST
and write the result out in a csv file, using pipe character | as the delimiter.

An example output can be seen in the "example.csv" file.
'''

SOLUTION

```
import xlrd
import os
import csv
from zipfile import ZipFile

datafile = "2013_ERCOT_Hourly_Load_Data.xls"
outfile = "2013_Max_Loads.csv"


def open_zip(datafile):
    with ZipFile('{0}.zip'.format(datafile), 'r') as myzip:
        myzip.extractall()
```

```
def parse_file(datafile):
    workbook = xlrd.open_workbook(datafile)
    sheet = workbook.sheet_by_index(0)
    data = []

    # YOUR CODE HERE
    # Remember that you can use xlrd.xldate_as_tuple(sometime, 0) to convert
    # Excel date to Python tuple of (year, month, day, hour, minute, second)
    data.append(['Station', 'Year', 'Month', 'Day', 'Hour', 'Max Load'])
    stations = sheet.row_values(0, start_colx=1, end_colx=9)
    max_load = 0
    for i in range(len(stations)):
        station_values = sheet.col_values(i + 1, start_rowx=1)

        max_row = station_values.index(max(station_values)) + 1
        Year, Month, Day, Hour, Minute, Second =
xlrd.xldate_as_tuple(sheet.cell_value(max_row,0), 0)
        data.append([stations[i], Year, Month, Day, Hour,
max(station_values)])

    #return data
    return data

def save_file(data, filename):
    with open( filename, 'wb') as csvfile:
        spamwriter = csv.writer(csvfile, delimiter='|')
        count = 0
        for i in data:
            print count
            count = count + 1
            spamwriter.writerow(i)
    # YOUR CODE HERE


def test():
    open_zip(datafile)
    data = parse_file(datafile)
    save_file(data, outfile)

    number_of_rows = 0
    stations = []

    ans = {'FAR_WEST': {'Max Load': '2281.2722140000024',
                        'Year': '2013',
                        'Month': '6',
                        'Day': '26',
                        'Hour': '17'}}
    correct_stations = ['COAST', 'EAST', 'FAR_WEST', 'NORTH',
```

```
                    'NORTH_C', 'SOUTHERN', 'SOUTH_C', 'WEST']
    fields = ['Year', 'Month', 'Day', 'Hour', 'Max Load']

    with open(outfile) as of:
        csvfile = csv.DictReader(of, delimiter="|")
        for line in csvfile:
            station = line['Station']
            if station == 'FAR_WEST':
                for field in fields:
                    # Check if 'Max Load' is within .1 of answer
                    if field == 'Max Load':
                        max_answer = round(float(ans[station][field]), 1)
                        max_line = round(float(line[field]), 1)
                        assert max_answer == max_line

                    # Otherwise check for equality
                    else:
                        assert ans[station][field] == line[field]

            number_of_rows += 1
            stations.append(station)

        # Output should be 8 lines not including header
        assert number_of_rows == 8

        # Check Station Names
        assert set(stations) == set(correct_stations)


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

Quiz 3 - Wrangling Json

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
This exercise shows some important concepts that you should be aware about:
- using codecs module to write unicode files
- using authentication with web APIs
- using offset when accessing web APIs

To run this code locally you have to register at the NYTimes developer site
and get your own API key. You will be able to complete this exercise in our UI
without doing so, as we have provided a sample result. (See the file
'popular-viewed-1.json' from the tabs above.)

Your task is to modify the article_overview() function to process the saved file that represents the most popular articles (by view count) from the last day, and return a tuple of variables containing the following data:
- labels: list of dictionaries, where the keys are the "section" values and values are the "title" values for each of the retrieved articles.
- urls: list of URLs for all 'media' entries with "format": "Standard Thumbnail"

All your changes should be in the article_overview() function. See the test() function for examples of the elements of the output lists.
The rest of functions are provided for your convenience, if you want to access the API by yourself.
"""

```
import json
import codecs
import requests

URL_MAIN = "http://api.nytimes.com/svc/"
URL_POPULAR = URL_MAIN + "mostpopular/v2/"
API_KEY = { "popular": "",
            "article": ""}


def get_from_file(kind, period):
    filename = "popular-{0}-{1}.json".format(kind, period)
    with open(filename, "r") as f:
        return json.loads(f.read())


def article_overview(kind, period):
    data = get_from_file(kind, period)
    titles = []
    urls =[]
    # YOUR CODE HERE
    k = 0
    for article_iterator in data:
        titles.append({article_iterator["section"] :
article_iterator["title"]})

        c = 0
        d = 0
        for media_iterator in article_iterator["media"]:
            c = c + 1

            for meta in media_iterator["media-metadata"]:
                d = d + 1
                k = k + 1
```

```
                if meta["format"] == "Standard Thumbnail":
                    c = c - 1
                    k = k + 1
                    t= len(urls)
                    p = meta["url"]
                    urls.insert(t, p)

    return (titles, urls)



def query_site(url, target, offset):
    # This will set up the query with the API key and offset
    # Web services often use offset paramter to return data in small chunks
    # NYTimes returns 20 articles per request, if you want the next 20
    # You have to provide the offset parameter
    if API_KEY["popular"] == "" or API_KEY["article"] == "":
        print "You need to register for NYTimes Developer account to run this
program."
        print "See Intructor notes for information"
        return False
    params = {"api-key": API_KEY[target], "offset": offset}
    r = requests.get(url, params = params)

    if r.status_code == requests.codes.ok:
        return r.json()
    else:
        r.raise_for_status()


def get_popular(url, kind, days, section="all-sections", offset=0):
    # This function will construct the query according to the requirements of
the site
    # and return the data, or print an error message if called incorrectly
    if days not in [1,7,30]:
        print "Time period can be 1,7, 30 days only"
        return False
    if kind not in ["viewed", "shared", "emailed"]:
        print "kind can be only one of viewed/shared/emailed"
        return False

    url += "most{0}/{1}/{2}.json".format(kind, section, days)
    data = query_site(url, "popular", offset)

    return data
```

```
def save_file(kind, period):
    # This will process all results, by calling the API repeatedly with
supplied offset value,
    # combine the data and then write all results in a file.
    data = get_popular(URL_POPULAR, "viewed", 1)
    num_results = data["num_results"]
    full_data = []
    with codecs.open("popular-{0}-{1}.json".format(kind, period),
encoding='utf-8', mode='w') as v:
        for offset in range(0, num_results, 20):
            data = get_popular(URL_POPULAR, kind, period, offset=offset)
            full_data += data["results"]

        v.write(json.dumps(full_data, indent=2))


def test():
    titles, urls = article_overview("viewed", 1)
    assert len(titles) == 20
    assert len(urls) == 30
    assert titles[2] == {'Opinion': 'Professors, We Need You!'}
    assert urls[20] ==
'http://graphics8.nytimes.com/images/2014/02/17/sports/ICEDANCE/ICEDANCE-thumb
Standard.jpg'


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

Problem Set 2 - Data in More Complex Format

Quiz 1 - Carrier List

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
Your task in this exercise is to modify 'extract_carrier()` to get a list of all airlines. Exclude all of the combination values like "All U.S. Carriers" from the data that you return. You should return a list of codes for the carriers.

All your changes should be in the 'extract_carrier()' function. The 'options.html' file in the tab above is a stripped down version of what is actually on the website, but should provide an example of what you should get

from the full file.

Please note that the function 'make_request()' is provided for your reference only. You will not be able to to actually use it from within the Udacity web UI.
"""

SOLUTION

```
from bs4 import BeautifulSoup
html_page = "options.html"


def extract_carriers(page):
    data = []
    with open(page, "r") as html:
        # do something here to find the necessary values
        soup = BeautifulSoup(html, "lxml")
        mydict = soup.find(id="CarrierList")
        #mydict2 = soup.find(id="__EVENTVALIDATION")
        print mydict
        mylist = mydict.find_all('option')
        print mylist
        print len(mylist)
        for i in range(3,len(mylist)):
          a = mylist[i]["value"]
          data.append(a)
        #mydata = soup.find("AirTran Airways")
        #print mydata
    #print data
    #print len(data)
    return data


def make_request(data):
    eventvalidation = data["eventvalidation"]
    viewstate = data["viewstate"]
    airport = data["airport"]
    carrier = data["carrier"]

    r = s.post("https://www.transtats.bts.gov/Data_Elements.aspx?Data=2",
               data = (("__EVENTTARGET", ""),
                       ("__EVENTARGUMENT", ""),
                       ("__VIEWSTATE", viewstate),
                       ("__VIEWSTATEGENERATOR",viewstategenerator),
                       ("__EVENTVALIDATION", eventvalidation),
                       ("CarrierList", carrier),
                       ("AirportList", airport),
                       ("Submit", "Submit")))
```

```
    return r.text


def test():
    data = extract_carriers(html_page)
    assert len(data) == 16
    assert "FL" in data
    assert "NK" in data

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

Quiz 2 - Airport List

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
Complete the 'extract_airports()' function so that it returns a list of airport codes, excluding any combinations like "All".

Refer to the 'options.html' file in the tab above for a stripped down version of what is actually on the website. The test() assertions are based on the given file.
"""

SOLUTION

```
from bs4 import BeautifulSoup
html_page = "options.html"


def extract_airports(page):
    data = []
    with open(page, "r") as html:
        # do something here to find the necessary values
        soup = BeautifulSoup(html, "lxml")
        mydict = soup.find(id="AirportList")
        mylist = mydict.find_all('option')
        for i in range(len(mylist)):
            a = mylist[i]["value"]
            if a[:3] != 'All':
                data.append(a)
            #print mydict
    #print data
    return data


def test():
    data = extract_airports(html_page)
```

```
    assert len(data) == 15
    assert "ATL" in data
    assert "ABR" in data

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

Quiz 3 - Processing All

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Let's assume that you combined the code from the previous 2 exercises with code
from the lesson on how to build requests, and downloaded all the data locally.
The files are in a directory "data", named after the carrier and airport:
"{}-{}.html".format(carrier, airport), for example "FL-ATL.html".

The table with flight info has a table class="dataTDRight". Your task is to
use 'process_file()' to extract the flight data from that table as a list of
dictionaries, each dictionary containing relevant data from the file and table
row. This is an example of the data structure you should return:

data = [{"courier": "FL",
        "airport": "ATL",
        "year": 2012,
        "month": 12,
        "flights": {"domestic": 100,
                    "international": 100}
       },
        {"courier": "..."}
]

Note - year, month, and the flight data should be integers.
You should skip the rows that contain the TOTAL data for a year.

There are couple of helper functions to deal with the data files.
Please do not change them for grading purposes.
All your changes should be in the 'process_file()' function.

The 'data/FL-ATL.html' file in the tab above is only a part of the full data,
covering data through 2003. The test() code will be run on the full table, but
the given file should provide an example of what you will get.
"""
```

SOLUTION

```
from bs4 import BeautifulSoup
from zipfile import ZipFile
import os

datadir = "data"


def open_zip(datadir):
    with ZipFile('{0}.zip'.format(datadir), 'r') as myzip:
        myzip.extractall()


def process_all(datadir):
    files = os.listdir(datadir)
    return files


def process_file(f):
    """
    This function extracts data from the file given as the function argument
in
    a list of dictionaries. This is example of the data structure you should
    return:

    data = [{"courier": "FL",
             "airport": "ATL",
             "year": 2012,
             "month": 12,
             "flights": {"domestic": 100,
                         "international": 100}
            },
            {"courier": "..."}
    ]


    Note - year, month, and the flight data should be integers.
    You should skip the rows that contain the TOTAL data for a year.
    """
    data = []
    info = {}
    #print f[:6]
    info["courier"], info["airport"] = f[:6].split("-")
    # Note: create a new dictionary for each entry in the output data list.
    # If you use the info dictionary defined here each element in the list
    # will be a reference to the same info dictionary.
    #print info
    with open("{}/{}".format(datadir, f), "r") as html:
```

```
        soup = BeautifulSoup(html,"lxml")
        a=0
        table = soup.find('table',class_="dataTDRight").find_all('tr',
class_='dataTDRight')
        for i in table:
            cols = i.find_all('td')
            print cols[1].text.lower().find('total')
            print cols[0].text
            b=cols[0].text
            c=cols[1].text
            print cols[1].text
            if cols[1].text.lower().find('total') == -1:
                info["year"] = int(b)
                a=a+1
                info["month"] = int(c)
                info["flights"] = {"domestic" : int(cols[2].text.replace(',',
''')),
                                   "international" :
int(cols[3].text.replace(',', ''))}

                #info = info.uppercase()
                data.append(info)

        #print table
    #print data
    return data



def test():
    print "Running a simple test..."
    open_zip(datadir)
    files = process_all(datadir)
    data = []
    # Test will loop over three data files.
    for f in files:
        data += process_file(f)

    assert len(data) == 399  # Total number of rows
    for entry in data[:3]:
        assert type(entry["year"]) == int
        assert type(entry["month"]) == int
        assert type(entry["flights"]["domestic"]) == int
        assert len(entry["airport"]) == 3
        assert len(entry["courier"]) == 2
    assert data[0]["courier"] == 'FL'
    assert data[0]["month"] == 10
    assert data[-1]["airport"] == "ATL"
```

```
    assert data[-1]["flights"] == {'international': 108289, 'domestic':
701425}

    print "... success!"

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

Quiz 4 and Quiz 5 - Patent Database and Result of Parsing the data file

Please enter content of the line that is causing the error:

```
<?xml version="1.0" encoding="UTF-8"?>
```

What do you think is the problem?

```
same file consists of several xml schema
```

Quiz 6 - Processing Patents

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# So, the problem is that the gigantic file is actually not a valid XML, because
# it has several root elements, and XML declarations.
# It is, a matter of fact, a collection of a lot of concatenated XML documents.
# So, one solution would be to split the file into separate documents,
# so that you can process the resulting files as valid XML documents.

import xml.etree.ElementTree as ET
PATENTS = 'patent.data'

def get_root(fname):
    tree = ET.parse(fname)
    return tree.getroot()


def split_file(filename):
    """
    Split the input file into separate files, each containing a single patent.
```

```
    As a hint - each patent declaration starts with the same line that was
    causing the error found in the previous exercises.

    The new files should be saved with filename in the following format:
    "{}-{}".format(filename, n) where n is a counter, starting from 0.
    """
    save_file_format = "{}-{}"
    patent_file = '<?xml version="1.0" encoding="UTF-8"?>'
    with open(filename, 'r') as file_reader:
        fileCounter = 0
         a=0
        file_writer = open(save_file_format.format(filename, fileCounter),
'w')
        file_writer.write(file_reader.next())

        for lines in file_reader:
            if lines.find(patent_file) > -1:
                print "Files found"
                file_writer.close()
                  a = a-1
                fileCounter += 1
                file_writer = open(save_file_format.format(filename,
fileCounter), 'w')

            file_writer.write(lines)

        file_writer.close()
    pass


def test():
    split_file(PATENTS)
    for n in range(4):
        try:
            fname = "{}-{}".format(PATENTS, n)
            f = open(fname, "r")
            if not f.readline().startswith("<?xml"):
                print "You have not split the file {} in the correct
boundary!".format(fname)
            f.close()
        except:
            print "Could not find file {}. Check if the filename is
correct!".format(fname)


test()
```

Problem Set 3 - Data Quality

Quiz 1 - Auditing Data Types

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
In this problem set you work with cities infobox data, audit it, come up with a cleaning idea and then clean it up. In the first exercise we want you to audit the datatypes that can be found in some particular fields in the dataset.
The possible types of values can be:
- NoneType if the value is a string "NULL" or an empty string ""
- list, if the value starts with "{"
- int, if the value can be cast to int
- float, if the value can be cast to float, but CANNOT be cast to int.
  For example, '3.23e+07' should be considered a float because it can be cast as float but int('3.23e+07') will throw a ValueError
- 'str', for all other values

The audit_file function should return a dictionary containing fieldnames and a SET of the types that can be found in the field. e.g.
{"field1": set([type(float()), type(int()), type(str())]),
 "field2": set([type(str())]),
  ....
}
The type() function returns a type object describing the argument given to the function. You can also use examples of objects to create type objects, e.g. type(1.1) for a float: see the test function below for examples.

Note that the first three rows (after the header row) in the cities.csv file are not actual data points. The contents of these rows should note be included when processing data types. Be sure to include functionality in your code to skip over or detect these rows.
"""

```
import codecs
import csv
import json
import pprint

CITIES = 'cities.csv'
```

```
FIELDS = ["name", "timeZone_label", "utcOffset", "homepage",
"governmentType_label",
          "isPartOf_label", "areaCode", "populationTotal",
"elevation",
          "maximumElevation", "minimumElevation", "populationDensity",
          "wgs84_pos#lat", "wgs84_pos#long", "areaLand", "areaMetro",
"areaUrban"]

def myInt(value):
    try:
        int(value)
        return 1
    except:
        return 0

def myFloat(value):
    try:
        float(value)
        return 1
    except:
        return 0
def audit_file(filename, fields):
    fieldtypes = {}
    for f in fields:
        fieldtypes[f] = set()

    with open(filename, 'r') as input:
        reader = csv.DictReader(input)

        for r in reader:
            a = r['URI']
            count = 0
            for i in range(len(a)):
                if a[i] == 'dbpedia':
                    count = count +  1
            if count == 0:
                count = -1

            if count > -1:

                for f in fields:
                    myField = fieldtypes[f]
```

```
                    if r[f] == "NULL" or row[f] == "":
                        myField = list(myField)
                        myField.append(type(None))
                        myField = set(myField)
                    elif r[f][0] == "{":
                        myField = list(myField)
                        myField.append(type([]))
                        myField = set(myField)
                    elif myInt(r[f]) == 1:
                        myField = list(myField)
                        myField.append(type(1))
                        myField = set(myField)
                    elif isFloat(r[f]) == 1:
                        myField = list(myField)
                        myField.append(type(1.1))
                        myField = set(myField)
                    else:
                        myField = list(myField)
                        myField.append(type(''))
                        myField = set(myField)
                fieldtypes = myField

    # YOUR CODE HERE
    return fieldtypes


def test():
    fieldtypes = audit_file(CITIES, FIELDS)

    pprint.pprint(fieldtypes)
    print fieldtypes["areaLand"]
    assert fieldtypes["areaLand"] == set([type(1.1), type([]),
type(None)])
    assert fieldtypes['areaMetro'] == set([type(1.1), type(None)])

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

Quiz 2 - Making a Choice

If you look at the previous exercise, you see that “areaLand” sometimes contains an array of 2 slightly different values. That really does not make much sense, since an area of a city should be a single value. So, we should assure that it is the case in our dataset. However we would have to make a choice of which value to keep. Which of the following do you think is the best** choice:

- Keep the first value
- Keep the second value
- (•) Keep the value with more significant digits
- Keep the highest value

***note that this is a bit ambiguous, feel free to discuss it on discussion forum*

Reason - High significant digits would mean more information.

Quiz 3 - Fixing the Area

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
In this problem set you work with cities infobox data, audit it, come up with a cleaning idea and then clean it up.

Since in the previous quiz you made a decision on which value to keep for the "areaLand" field, you now know what has to be done.

Finish the function fix_area(). It will receive a string as an input, and it has to return a float representing the value of the area or None.
You have to change the function fix_area. You can use extra functions if you like, but changes to process_file will not be taken into account.
The rest of the code is just an example on how this function can be used.
"""

SOLUTION

```
import codecs
import csv
import json
import pprint
import re

CITIES = 'cities.csv'
```

```
def fix_area(area):
    myArea = area
    check1 = myArea[0]
    check2 = myArea[:-1]
    if check1 =='{':

        a = re.sub('{', '', myArea)
        b = re.sub('}', '', a)
        myList = b.split('|')
        ar1 = myList[0]
        ar2 = myList[1]
        l1 = len(ar1)
        l2 = len(ar2)
        print l1
        print l2
        if l1 > l2 :
            myArea = float(ar1)
        else:
            myArea = float(ar2)
    elif myArea == 'NULL' or '':
        myArea = None
    else:
        myArea = float(area)
    print myArea
    area = myArea
    #return area

    # YOUR CODE HERE

    return area



def process_file(filename):
    # CHANGES TO THIS FUNCTION WILL BE IGNORED WHEN YOU SUBMIT THE EXERCISE
    data = []

    with open(filename, "r") as f:
        reader = csv.DictReader(f)

        #skipping the extra metadata
        for i in range(3):
            l = reader.next()

        # processing file
        for line in reader:
            # calling your function to fix the area value
```

```
            if "areaLand" in line:
                line["areaLand"] = fix_area(line["areaLand"])
            data.append(line)

    return data


def test():
    data = process_file(CITIES)

    print "Printing three example results:"
    for n in range(5,8):
        pprint.pprint(data[n]["areaLand"])

    assert data[3]["areaLand"] == None
    print data[8]["areaLand"]
    assert data[8]["areaLand"] == 55166700.0
    assert data[20]["areaLand"] == 14581600.0
    assert data[33]["areaLand"] == 20564500.0


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

Quiz 4 - Other Fields

Explore the data and mark the fields that you think also should be processed in a similar way as “areaLand” (changed from an array to a single value):

- [ ] name
- [x] populationTotal
- [x] areaMetro
- [ ] postalCode

Reason - areaLand is similar to populationTotal and areaMetro. Name and postalCode are very different from areaLand.

Quiz 5 - Fixing Name

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
In this problem set you work with cities infobox data, audit it, come up with a cleaning idea and then clean it up.

In the previous quiz you recognized that the "name" value can be an array (or list in Python terms). It would make it easier to process and query the data later if all values for the name are in a Python list, instead of being just a string separated with special characters, like now.

Finish the function fix_name(). It will recieve a string as an input, and it will return a list of all the names. If there is only one name, the list will have only one item in it; if the name is "NULL", the list should be empty. The rest of the code is just an example on how this function can be used.
"""

SOLUTION

```
import codecs
import csv
import pprint
import re

CITIES = 'cities.csv'


def fix_name(name):

    # YOUR CODE HERE
    myName = name
    check1 = myName[0]
    check2 = myName[:-1]
    count = 0
    i = 0
    if check1 == '{':
        a = re.sub('{', '', myName)
        b = re.sub('}', '', a)
        myName = b.split('|')
        count = count + 1
    elif myName == 'NULL':
        myName = []
        i = i + 1
    else:
        myName = [myName]
    name = myName
    print name
    return name
```

```
def process_file(filename):
    data = []
    with open(filename, "r") as f:
        reader = csv.DictReader(f)
        #skipping the extra metadata
        for i in range(3):
            l = reader.next()
        # processing file
        for line in reader:
            # calling your function to fix the area value
            if "name" in line:
                line["name"] = fix_name(line["name"])
            data.append(line)
    return data


def test():
    data = process_file(CITIES)

    print "Printing 20 results:"
    for n in range(20):
        pprint.pprint(data[n]["name"])

    assert data[14]["name"] == ['Negtemiut', 'Nightmute']
    assert data[9]["name"] == ['Pell City Alabama']
    assert data[3]["name"] == ['Kumhari']

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

Quiz 6 - Crossfield Auditing

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
In this problem set you work with cities infobox data, audit it, come up with a cleaning idea and then clean it up.

If you look at the full city data, you will notice that there are couple of values that seem to provide the same information in different formats: "point" seems to be the combination of "wgs84_pos#lat" and "wgs84_pos#long". However, we do not know if that is the case and should check if they are equivalent.

Finish the function check_loc(). It will recieve 3 strings: first, the combined value of "point" followed by the separate "wgs84_pos#" values. You have to

extract the lat and long values from the "point" argument and compare them to the "wgs84_pos# values, returning True or False.

Note that you do not have to fix the values, only determine if they are consistent. To fix them in this case you would need more information. Feel free to discuss possible strategies for fixing this on the discussion forum.

The rest of the code is just an example on how this function can be used. Changes to "process_file" function will not be taken into account for grading.
"""

SOLUTION

```
import csv
import pprint

CITIES = 'cities.csv'


def check_loc2(point, lat, longi):
    # YOUR CODE HERE
    a = point
    p = len(a)
    ct = 0
    myList = []
    while ct!=p:
        print ct
        if a!='NULL' and a[ct]==' ':
            print ct
            print myList
            myList[0] = a[:ct]
            myList[1] = a[ct:]
            break
        else:
            ct = ct+1
    if myList[0]==lat and myList[1]==longi:
        return True
    else:
        return False
    #pass

def process_file(filename):
    data = []
    with open(filename, "r") as f:
        reader = csv.DictReader(f)
        #skipping the extra matadata
        for i in range(3):
            l = reader.next()
```

```
        # processing file
        for line in reader:
            # calling your function to check the location
            result = check_loc(line["point"], line["wgs84_pos#lat"],
line["wgs84_pos#long"])
            if not result:
                print "{}: {} != {} {}".format(line["name"], line["point"],
line["wgs84_pos#lat"], line["wgs84_pos#long"])
            data.append(line)

    return data


def test():
    assert check_loc("33.08 75.28", "33.08", "75.28") == True
    assert check_loc("44.57833333333333 -91.21833333333333", "44.5783",
"-91.2183") == False

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

Quiz 7 - SQL or MongoDB

Chose MongoDB

Problem Set 4- Working with MongoDB

Quiz 1- Preparing Data

#!/usr/bin/env python
-*- coding: utf-8 -*-
"""
In this problem set you work with another type of infobox data, audit it,
clean it, come up with a data model, insert it into MongoDB and then run some
queries against your database. The set contains data about Arachnid class
animals.

Your task in this exercise is to parse the file, process only the fields that
are listed in the FIELDS dictionary as keys, and return a list of dictionaries
of cleaned values.

The following things should be done:
- keys of the dictionary changed according to the mapping in FIELDS dictionary
- trim out redundant description in parenthesis from the 'rdf-schema#label'
  field, like "(spider)"
- if 'name' is "NULL" or contains non-alphanumeric characters, set it to the

same value as 'label'.
- if a value of a field is "NULL", convert it to None
- if there is a value in 'synonym', it should be converted to an array (list)
by stripping the "{}" characters and splitting the string on "|". Rest of the
cleanup is up to you, e.g. removing "*" prefixes etc. If there is a singular
synonym, the value should still be formatted in a list.
- strip leading and ending whitespace from all fields, if there is any
- the output structure should be as follows:

```
[ { 'label': 'Argiope',
    'uri': 'http://dbpedia.org/resource/Argiope_(spider)',
    'description': 'The genus Argiope includes rather large and spectacular spiders that often ...',
    'name': 'Argiope',
    'synonym': ["One", "Two"],
    'classification': {
                  'family': 'Orb-weaver spider',
                  'class': 'Arachnid',
                  'phylum': 'Arthropod',
                  'order': 'Spider',
                  'kingdom': 'Animal',
                  'genus': None
                  }
  },
  { 'label': ... , }, ...
]
```

* Note that the value associated with the classification key is a dictionary
with taxonomic labels.
"""

SOLUTION

```
import codecs
import csv
import json
import pprint
import re

DATAFILE = 'arachnid.csv'
FIELDS ={'rdf-schema#label': 'label',
         'URI': 'uri',
         'rdf-schema#comment': 'description',
         'synonym': 'synonym',
         'name': 'name',
         'family_label': 'family',
         'class_label': 'class',
```

```
          'phylum_label': 'phylum',
          'order_label': 'order',
          'kingdom_label': 'kingdom',
          'genus_label': 'genus'}


def process_file(filename, fields):

    process_fields = fields.keys()
    data = []
    with open(filename, "r") as f:
        reader = csv.DictReader(f)
        for i in range(3):
            l = reader.next()

        for line in reader:
            if line['synonym']!=None:
                my_old = line['synonym']
                myString = my_old[1:-1]
                line['synonyn'] = parse_array(my_old)
                for s in line['synonym']:
                    s.replace('*', "")
                print line['synonyn']

            if line['rdf-schema#label']!='NULL':
                myOld = line['rdf-schema#label']
                myString = myOld[1:-1]
                line['rdf-schema#label'] = myString
            #myString = MongoDBreturned[1:-1]

            if line['name']=='NULL' or line['name'].isalpha():
                line['name'] = line['rdf-schema#label']



            for p in process_fields:
                if line[p]=='NULL':
                    line[p]=None
                else:
                    line[p].strip()
            # YOUR CODE HERE
            club_data = {}
            club_data['classification'] = {}
            for j in fields:
                if re.search(r'_label', j):
                    club_data['classification'][fields[j]] = line[j]
                else:
                    club_data[fields[j]] = line[j]
```

```
            data.append(club_data)

            pass
    return data


def parse_array(v):
    if (v[0] == "{") and (v[-1] == "}"):
        v = v.lstrip("{")
        v = v.rstrip("}")
        v_array = v.split("|")
        v_array = [i.strip() for i in v_array]
        return v_array
    return [v]


def test():
    data = process_file(DATAFILE, FIELDS)
    print "Your first entry:"
    pprint.pprint(data[0])
    first_entry = {
        "synonym": None,
        "name": "Argiope",
        "classification": {
            "kingdom": "Animal",
            "family": "Orb-weaver spider",
            "order": "Spider",
            "phylum": "Arthropod",
            "genus": None,
            "class": "Arachnid"
        },
        "uri": "http://dbpedia.org/resource/Argiope_(spider)",
        "label": "Argiope",
        "description": "The genus Argiope includes rather large and
spectacular spiders that often have a strikingly coloured abdomen. These
spiders are distributed throughout the world. Most countries in tropical or
temperate climates host one or more species that are similar in appearance.
The etymology of the name is from a Greek name meaning silver-faced."
    }

    assert len(data) == 76
    assert data[0] == first_entry
    assert data[17]["name"] == "Ogdenia"
    assert data[48]["label"] == "Hydrachnidiae"
    assert data[14]["synonym"] == ["Cyrene Peckham & Peckham"]

if __name__ == "__main__":
```

```
    test()
```

Quiz 2 - Inserting into DB

"""

Complete the insert_data function to insert the data into MongoDB.

"""

SOLUTION

```
import json

def insert_data(data, db):
    my_data = data
    #collect =
    db.arachnid.insert(my_data)

    # Your code here. Insert the data into a collection 'arachnid'

    #pass


if __name__ == "__main__":

    from pymongo import MongoClient
    client = MongoClient("mongodb://localhost:27017")
    db = client.examples

    with open('arachnid.json') as f:
        data = json.loads(f.read())
        insert_data(data, db)
        print db.arachnid.find_one()
```

Problem Set 5 - Analyzing Data

Quiz 1 - Most Common City Name

#!/usr/bin/env python

"""

Use an aggregation query to answer the following question.

What is the most common city name in our cities collection?

Your first attempt probably identified None as the most frequently occurring city name. What that actually means is that there are a number of cities without a name field at all. It's strange that such documents would exist in this collection and, depending on your situation, might actually warrant

further cleaning.

To solve this problem the right way, we should really ignore cities that don't have a name specified. As a hint ask yourself what pipeline operator allows us to simply filter input? How do we test for the existence of a field?

Please modify only the 'make_pipeline' function so that it creates and returns an aggregation pipeline that can be passed to the MongoDB aggregate function. As in our examples in this lesson, the aggregation pipeline should be a list of one or more dictionary objects. Please review the lesson examples if you are unsure of the syntax.

Your code will be run against a MongoDB instance that we have provided. If you want to run this code locally on your machine, you have to install MongoDB, download and insert the dataset. For instructions related to MongoDB setup and datasets please see Course Materials.

Please note that the dataset you are using here is a different version of the cities collection provided in the course materials. If you attempt some of the same queries that we look at in the problem set, your results may be different.
"""

SOLUTION

```
def get_db(db_name):
    from pymongo import MongoClient
    client = MongoClient('localhost:27017')
    db = client[db_name]
    return db

def make_pipeline():
    # complete the aggregation pipeline
    pipeline = [ ]
    ct = 0
    myDict1 = {'$match': {'name': {'$exists': 1}}}
    myDict2 = {'$group': {'_id': '$name',
                                'count': {'$sum':1}}}
    myDict3 = {'$sort': {'count': -1}}
    myDict4 = {'$limit':1}
    pipeline.append(myDict1)
    ct = ct + 1
    pipeline.append(myDict2)
    ct = ct + 1
    pipeline.append(myDict3)
    a = len(pipeline)
    print a
    pipeline.append(myDict4)
```

```
    print pipeline
    return pipeline

def aggregate(db, pipeline):
    return [doc for doc in db.cities.aggregate(pipeline)]


if __name__ == '__main__':
    # The following statements will be used to test your code by the grader.
    # Any modifications to the code past this point will not be reflected by
    # the Test Run.
    db = get_db('examples')
    pipeline = make_pipeline()
    result = aggregate(db, pipeline)
    import pprint
    pprint.pprint(result[0])
    assert len(result) == 1
    assert result[0] == {'_id': 'Shahpur', 'count': 6}
```

Quiz 2 - Region Cities

#!/usr/bin/env python
"""
Use an aggregation query to answer the following question.

Which Region in India has the largest number of cities with longitude between
75 and 80?

Please modify only the 'make_pipeline' function so that it creates and returns
an aggregation pipeline that can be passed to the MongoDB aggregate function.
As in our examples in this lesson, the aggregation pipeline should be a list of
one or more dictionary objects. Please review the lesson examples if you are
unsure of the syntax.

Your code will be run against a MongoDB instance that we have provided. If you
want to run this code locally on your machine, you have to install MongoDB,
download and insert the dataset. For instructions related to MongoDB setup and
datasets please see Course Materials.

Please note that the dataset you are using here is a different version of the
cities collection provided in the course materials. If you attempt some of the
same queries that we look at in the problem set, your results may be different.
"""

SOLUTION

```
def get_db(db_name):
    from pymongo import MongoClient
```

```
    client = MongoClient('localhost:27017')
    db = client[db_name]
    return db

def make_pipeline():
    # complete the aggregation pipeline
    pipeline = [ ]
    myDict1 = {'$match': {'country': 'India',
                          'lon': {'$gte': 75,
                                  '$lte': 80}}}
    myDict2 = {'$unwind': '$isPartOf'}
    myDict3 = {'$group': {'_id': '$isPartOf',
                          'count': {'$sum': 1}}}
    myDict4 = {'$sort': {'count': -1}}
    myDict5 = {'$limit': 1}
    pipeline.append(myDict1)
    pipeline.append(myDict2)
    a = len(pipeline)
    print a
    pipeline.append(myDict3)
    print pipeline
    pipeline.append(myDict4)
    a = len(pipeline)
    print a
    pipeline.append(myDict5)
    return pipeline

def aggregate(db, pipeline):
    return [doc for doc in db.cities.aggregate(pipeline)]

if __name__ == '__main__':
    # The following statements will be used to test your code by the grader.
    # Any modifications to the code past this point will not be reflected by
    # the Test Run.
    db = get_db('examples')
    pipeline = make_pipeline()
    result = aggregate(db, pipeline)
    import pprint
    pprint.pprint(result[0])
    assert len(result) == 1
    assert result[0]["_id"] == 'Tamil Nadu'
    assert result[0]["count"] == 424
```

Quiz 3 - Average Population

#!/usr/bin/env python
"""
Use an aggregation query to answer the following question.

Extrapolating from an earlier exercise in this lesson, find the average regional city population for all countries in the cities collection. What we are asking here is that you first calculate the average city population for each region in a country and then calculate the average of all the regional averages for a country.

As a hint, _id fields in group stages need not be single values. They can also be compound keys (documents composed of multiple fields). You will use the same aggregation operator in more than one stage in writing this aggregation query. I encourage you to write it one stage at a time and test after writing each stage.

Please modify only the 'make_pipeline' function so that it creates and returns an aggregation pipeline that can be passed to the MongoDB aggregate function. As in our examples in this lesson, the aggregation pipeline should be a list of one or more dictionary objects. Please review the lesson examples if you are unsure of the syntax.

Your code will be run against a MongoDB instance that we have provided. If you want to run this code locally on your machine, you have to install MongoDB, download and insert the dataset. For instructions related to MongoDB setup and datasets please see Course Materials.

Please note that the dataset you are using here is a different version of the cities collection provided in the course materials. If you attempt some of the same queries that we look at in the problem set, your results may be different.
"""

SOLUTION

```
def get_db(db_name):
    from pymongo import MongoClient
    client = MongoClient('localhost:27017')
    db = client[db_name]
    return db

def make_pipeline():
    # complete the aggregation pipeline
    pipeline = [ ]
    myDict1 = {'$unwind': '$isPartOf'}
    myDict2 = {'$group': {'_id': {'country': '$country',
                                  'region': '$isPartOf'},
                          'avg': {'$avg': '$population'}}}
    myDict3 = {'$group': {'_id': '$_id.country',
                          'avgRegionalPopulation': {'$avg': '$avg'}}}
    pipeline.append(myDict1)
```

```
    #print pipeline
    a = len(pipeline)
    pipeline.append(myDict2)
    assert len(pipeline) == a + 1
    pipeline.append(myDict3)
    assert len(pipeline) == a + 2
    return pipeline

def aggregate(db, pipeline):
    return [doc for doc in db.cities.aggregate(pipeline)]

if __name__ == '__main__':
    # The following statements will be used to test your code by the grader.
    # Any modifications to the code past this point will not be reflected by
    # the Test Run.
    db = get_db('examples')
    pipeline = make_pipeline()
    result = aggregate(db, pipeline)
    import pprint
    if len(result) < 150:
        pprint.pprint(result)
    else:
        pprint.pprint(result[:100])
    key_pop = 0
    for country in result:
        if country["_id"] == 'Lithuania':
            assert country["_id"] == 'Lithuania'
            assert abs(country["avgRegionalPopulation"] - 14750.784447977203)
< 1e-10
            key_pop = country["avgRegionalPopulation"]
    assert {'_id': 'Lithuania', 'avgRegionalPopulation': key_pop} in result
```