Overview: Write a Python program to analyze data about artists, users, play counts, and tags from last.fm.

Learning objectives: Gain experience processing and cleaning datasets, building data structures to support analysis, and performing data analytics to answer specific questions.

Project specification:
For this project, you will analyze data from the on-line music service, last.fm. Specifically, we will use a dataset that was released as part of the HetRec Workshop at the 2011 ACM Conference on Recommender Systems. A zip file containing the data is made available by the Grouplens research team at the University of Minnesota. To download it, go to the page: http://grouplens.org/datasets/hetrec-2011/
and download the file: hetrec2011-lastfm-2k.zip

Unzip the file and you should get a folder with 7 items:

Filename	Notes
artists.dat	This file contains artist ids, artist names, and other information such as URLs for artists on the last.fm site.
readme.txt	Important notes about the dataset and file structures.
user_artists.dat	This file contains lines that describe how many times a user has played songs by an artist. It contains lines with user ids, artist ids, and the number of times the user has played a song by that artist.
user_friends.dat	Contains lines with a one-directional friend relations. For example, a line with "2 275" indicates that uid 2 is friends with uid 275. If 275 is friends with 2, then there will be a second line with "275 2".
user_taggedartists.dat	Contains lines with user id, artist id, tag id, day, month, and year. Each line indicates that the user made a tag for the given artist on that day/month/year. We will not analyze the contents of the tags (in the tags.dat file), but instead will focus on when and how many tags were made by users for particular artists.
tags.dat	Do not use this file for this project.
user_taggedartists_timestamps.dat	Do not use this file for this project.

This dataset is just a sampling of data from last.fm, but it is a good example of the type of real-world data that data analysts work with. Here are a few stats about the data files: there are 1892 user ids and 17,632 artists. There are 92,834 user-listed_to-artist relationships, and 25,434 friend relationships.

Reading and Cleaning the Data
Your program should start by reading the data contained in the .dat files described above and storing the data in Python data structures. The .dat files are text files with fields separated by tab characters. The text is stored using "utf-8" encoding. To properly read utf-8 characters, you will need to use the "codecs" library:

```
import codecs
fp = codecs.open("artists.dat", encoding="utf-8")
fp.readline()  #skip first line of headers
for line in fp:
    line = line.strip()
    # do other processing
```

As with many real-world datasets, there are a few problems and inconsistencies with the data. I will point out a few of these and outline how you should address them. However, as you work with the dataset, you may find other issues. If you run into data problems and have questions, please feel free to contact me.

In the file user_taggedartists.dat there are a few records for tags that were made in the 1950s! Since this was well before last.fm existed, we will assume that these tag records are erroneous and will exclude them from our analysis. In the same file, there are records that indicate a tag was made for an artist id that does not exist in the artists.dat file. We will also exclude these from our tag data.

Storing and Indexing the Data
As you read the data from the .dat files, you will need to store it in Python data structures. In addition, to support the data analytics, you will need to build additional (sometimes complex) data structures. Later in the semester, we will learn about Python packages designed to store and manipulate datasets such as this. However, for this assignment, you should use only built-in Python data types (e.g., strings, lists, dicts, tuples) except as otherwise noted in this specification. Within those constraints, you have flexibility about how you store the data and what types of data structures you build. Below, I have outlined some approaches that worked well for me as starting points.

- Based on the artists.dat data, create an `aid2name` dictionary to map aids to the artist name. This will make printing out the artist names and interpreting the data much easier.

- Based on the user_taggedartists.dat file, create a list of dictionaries that stores records consisting of user id, artist id, tag id, day, month, and year. I abbreviated this as `uat` for short. For example, here are the first three items in my `uat` data structure:

```
[{'uid': 2, 'month': 4, 'year': 2009, 'tid': 13, 'aid': 52, 'day': 1}
 {'uid': 2, 'month': 4, 'year': 2009, 'tid': 15, 'aid': 52, 'day': 1}
 {'uid': 2, 'month': 4, 'year': 2009, 'tid': 18, 'aid': 52, 'day': 1}
 ...
]
```

  Don't put any "bad" data in `uat`. In other words, don't append any records with years<2000 or that have an artist id that is not in the artists.dat file.

- Based on the user_artists.dat file, create a data structure that stores records consisting of user id, artist id, and play count. Note that in the user_artists.dat file, the play count is called "weight". This number represents the number of times that user played songs by that artist. I called this `uap` for short. Only include artists that appear in the artists.dat file. Here are the first three lines of my `uap` data structure:

```
[{'aid': 51, 'uid': 2, 'weight': 13883}
 {'aid': 52, 'uid': 2, 'weight': 11690}
 {'aid': 53, 'uid': 2, 'weight': 11351}
 ...
]
```

- Use the data to create a set of dictionaries that map artist ids to computed counts of the total number of tags made for that artist, and the total number of play counts of the artists' songs. Create similar dictionaries that map users to the total number of tags they have made and the total number of song plays that they have made (across all artists' songs).

Data Analytics and Specific Queries

For the data analytics, you will answer a specific set of questions given below. For each question, clearly comment the section of your code that addresses the question, and in your comments, provide a brief description of the approach that you took. Implement functions as appropriate.

There is no "user interface" for this project. I will run your program and it should produce output all at once for the questions below.

In your output for each question, print:

- A blank line
- A line of 40 exclamation points "!!!!!!!!"
- Another blank line
- A line with the question number and the brief description of the question
- Another blank line
- Then print the output for that question.

1. *Who are the top artists?* Print a list of the 10 artists with the most song plays (across all users), sorted by number of song plays. For each of the top 10, print the artist name, the artist id, and the total number of song plays for that artist. See example output below.

```
1. Who are the top artists?
    Britney Spears(289) 2393140
    Depeche Mode(72) 1301308
    Lady Gaga(89) 1291387
```

2. *Who are the top users?* Print the 10 user ids with the most song plays (across all artists), sorted by number of song plays. For each, print the user id and total number of song plays for that user.

3. *What artists have the most listeners?* Print a list of the 10 artists with the highest number of users who have listened to at least one song by that artist, sorted by number of listeners. For each artist, print the artist name, artist id, and the number of distinct users who have listened to a song by that artist.

4. *What artists have the highest average number of plays per listener?* The previous question (#3) asked you to compute the artists with the most listeners, but only tells part of the story. If Person A only played a Britney Spears song once, and Person B played Britney Spears songs 10,000 times, they would both be counted equally in Question #3. For this question (#4), we want to compute the average number of plays per listener for each artist. This average is the number of total plays for that artist divided by the number of listeners for that artist. Print a list of the 10 artists with the highest average number of plays per listener. Include the artist name, artist id, total number of plays, total number of listeners, and the computed average number of plays per listener.

5. *What artists with at least 50 listeners have the highest average number of plays per listener?* When you finish question (#4), you may notice a problem – many of the "average number of plays" end up being based on the data from only *one* user. According to my output for question #4, there is only one person who listened to the artist "Viking Quest", but that person really liked them, playing their songs 35,323 times. Since 35,323 / 1 = 35,323, Viking Quest got the highest average number of plays. Our metric from question #4 did not take into account that there are many artists that only have a few people who listen to them. For this question (#5), we will fix this by requiring that artists have at least 50 listeners before we "trust" the average number of plays. For output, print the same information as from question #4.

6. *Do users with five or more friends listen to more songs?* To answer this question, compute the total number of song plays for all users who have five or more friends. Divide this by the total number of users who have five or more friends to arrive at an average number of song plays for these users. Do the same for the set of users who have less than five friends. Print both numbers with clear labels.

7. *How similar are two artists?* There are many ways you might define "similar artists". Artists could be similar because of their musical style, the era they performed during, or based on the fans they have in common. For this question, we will define similarity based on common listeners. The *Jaccard index* is a statistic for measuring the similarity of two sets (http://en.wikipedia.org/wiki/Jaccard_index). Write a function called `artist_sim(aid1, aid2)` that takes two artist ids as arguments. The function should first compute the set of users who have listened to aid1, and the set of users who have listened to aid2. Then it should compute the Jaccard index of these two sets. The Jaccard index is the number of items in the intersection divided by the number of items in the union of the two sets. You will need to use floating point numbers when computing the index, and you may also find it helpful to use Python's `set` data structure (http://docs.python.org/2/library/sets.html). Test your function by computing the similarity of the following pairs of artists. For each pair, print the names of the two artists followed by the Jaccard index.

```
artist_sim(735,562)
artist_sim(735,89)
artist_sim(735,289)
artist_sim(89,289)
artist_sim(89,67)
artist_sim(67,735)
```

8. *For each month in 2005, what artists were tagged the most?* For each month in 2005 in the dataset (Aug 2005 – Dec 2005), print a list of the 10 artists with the highest number of tags for that month. Structure your output to show the month and year on a line by themselves, followed by lines with the artist name, artist id, and number of tags for that artist for that month+year, as shown below.

```
Aug 2005
  David Bowie(599):  num tags = 11
  New Order(59):  num tags = 7
  Rod Stewart(2861):  num tags = 7
  Ultravox(1013):  num tags = 6
  Siouxsie and the Banshees(1083):  num tags = 6
  Duran Duran(51):  num tags = 5
  INXS(99):  num tags = 5
  The Beatles(227):  num tags = 5
  Heavenly(4324):  num tags = 5
  Lou Reed(4541):  num tags = 5

Sep 2005
  Blood Ruby(11892):  num tags = 13
  The Fleshtones(550):  num tags = 3
  Greg Dulli(14468):  num tags = 3
  ...
```

9. For the 10 artists with the highest overall number of tags, list: a) the first month they entered the top 10 in terms of number of tags, and b) the number of months they were in the top 10 in terms of number of tags. The list should be sorted in order of the overall number of tags (e.g., Britney Spears is first with 931 total tags). Example output is shown below.

```
Britney Spears(289):  num tags = 931
  first month in top10 = Sep 2006
  months in top10 =  25

Lady Gaga(89):  num tags = 767
  first month in top10 = Dec 2008
  months in top10 =  19
```

Grading:
Your program will be evaluated based on its functionality, programming logic, and programming style. Functionality focuses on the question, "Does your program product the correct results?" Programming logic considers whether the approach you implemented in your code is correct (or close to correct). Programming style looks at how easy it is to understand your code – is it organized well, did you use functions appropriately, did you include good comments?

How to turn in your assignment:
Your program should be contained in a single file and be entirely code that you write yourself. Name your file according to the following convention:

```
youronyen_p2.py
```

Replace *youronyen* with your actual Onyen (e.g. my assignment would be `rcapra_p2.py`). The character between *youronyen* and the "p2.py" part should be a single underscore. There should be no spaces or other characters in the filename. Files with names that do not follow this convention may receive a substantial grade deduction.

I will test your program by running it with Python 2.7, with the specified .dat files in the same directory.

Submit your file electronically through the Sakai by going to the Assignments area and finding the "P2" assignment. After you think you have submitted the assignment, I strongly recommend checking to be sure the file was uploaded correctly by clicking on it from within Sakai. Keep in mind that if I cannot access your file, I cannot grade it.

If for some reason you need to re-submit your file, you must add a version number to your filename. Sakai is configured so that it will accept up to 3 total submissions. Use the following file naming convention if you need to re-submit:

Your first submission:	`youronyen_p2.py`
Your second submission:	`youronyen_p2_v2.py`
Your third submission:	`youronyen_p2_v3.py`

Sakai is also configured with a due date and an "accept until" date. Submissions received after the due date (even just 1 minute!) will receive a 10% penalty per day. The "accept until" date is 5 days after the due date. Unless you have made arrangements with me in advance, submissions will not be accepted after the "accept until" date and will have a score of zero recorded.