**Introduction to Programming – Fall 2017**

**Assignment 2**

**Notes:**

1. You may discuss concepts and ideas for solving the problems with others, but everyone must submit their own programs (no group submissions). Feel free to discuss with Mark or me in case you get stuck, have doubts, need clarification or some high-level guidance to refine your approach.
2. In order for it to be considered for grading, the program you submit **\*\*must\*\*** run and produce output. For example, we should be able to go to the shell/Terminal or PyCharm and execute your program to see the results. Partial grading (in case of partially correct answers) will only be done if this condition is met.
3. E-mail an attached copy of your scripts or a link to a github repository containing your scripts.
4. **Due date:** 13 Oct 2017, by 11:59 pm.

**Question 1:**

A file called ‘cleaned\_GPMDB\_table.tsv.zip’ has been placed in the Assignments/a2 folder on the ITP github. Once extracted this file will be called ‘cleaned\_GPMDB\_table.tsv’. It is a tab-separated file, this means that each row contains a set of values which are separated with a ‘\t’ character.

Eg:

row1\_val1\trow1\_val2\trow1\_val3\t\n

row2\_val1\trow2\_val2\trow2\_val3\t\n

row3\_val1\trow3\_val2\trow3\_val3\t\n

Each row in this file summarizes an XML file used by GPMDB, which is an open proteomics database storing mass-spectrometry-based protein identification information.

The first row of the file contains the header information; The headers provide a description for each value in a row. The header values are also separated using a ‘\t’ character.

Here are all the values listed in the header row of the table:

0: file\_name 1: date

2: total\_spectra\_assigned 3: unique\_spectra\_assigned

4: spectra\_used 5: num\_u\_proteins

6: most\_common\_protein 7: mcp\_count

8: num\_peptides 9: num\_unique\_peptides

10: cleavage\_C\_terminal 11: cleavage\_N\_terminal

12: fragment\_monoisotopic\_me 13: fragment\_monoisotopic\_me\_unit

14: parent\_monoisotopic\_me\_minus 15: parent\_monoisotopic\_me\_plus

16: parent\_monoisotopic\_me\_unit 17: parent\_monoisotopic\_mie

18: b\_ions 19: y\_ions

20: z\_ions 21: c\_ions

22: taxon

**Your goal is to perform the following steps**:

**Step 1:** Read the tsv file and save the contents in a nested list of rows (each row is itself a list).

Eg:

[[row1\_val\_1, row1\_val\_2, …, row1\_val22],

[row2\_val\_1, row2\_val\_2, …, row2\_val22],

[row3\_val\_1, row3\_val\_2, …, row3\_val22],

…

]

**Step 2:** Using ‘***map***’ function, create a dictionary for each row. Remember that ‘***map***’ applies a function / transformation to each item (row, in this case) in an iterable (list of rows, in this case). The dictionary should have entries for the date, most\_common\_protein and taxon found in the row, i.e.

{‘date’: date\_from\_row, ‘taxon’: taxon\_from\_row, ‘mcp’: most\_common\_protein}

**Tip:** Python has a datetime module which can convert a string into a date type. Date type allows you to access the year, month, day individually.

**Tip:** This is a good place to split the taxon string (which contains multiple taxons) into a list of strings where each string is an individual taxon.

You should now have a **list of dictionaries**

Eg:

[{‘date’: row1\_date, ‘taxon’: row1\_taxon, ‘mcp’: row1\_most\_common\_protein},

{‘date’: row2\_date, ‘taxon’: row2\_taxon, ‘mcp’: row1\_most\_common\_protein},

{‘date’: row3\_date, ‘taxon’: row3\_taxon, ‘mcp’: row1\_most\_common\_protein},

…

]

**Step 3:**  Filter the list of dictionaries so that any rows with TRYP or BOVIN in the most common protein string are removed.

eg:

sp|TRYP\_PIG| should be removed

sp|ALBU\_BOVIN| should be removed

You should now have a **filtered** **list of dictionaries**

**Step 4:** Use a ‘***reduce***’ function to reduce the filtered list of dictionaries down to a single dictionary containing counts for each taxon grouped by year. Finally print the most common taxon for each year and how many times it appeared that year.

**Tip 1:** When you use ‘reduce’ you can specify an initial value, in this case it will be an empty dictionary {}.

**reduce(function\_to\_appy, filtered\_list\_of\_dictionaries, {})**

This initial dict is the accumulator which will hold the desired results of the reduce step.

**Tip 2**: Reduce applies a function that takes two arguments at a time. When initial value is supplied, it takes the initial value and the first value in the iterable as the two arguments at the first reduction step. At every subsequent step, it uses the result of previous reduction step and the next item in iterable as arguments, until all items in the iterable have been reduced. Seen the example below on how reduce operates when an initial value is supplied

list\_of\_characters = [‘**b**’, ‘**c**’, ‘**d**’]

def join\_characters(char\_1, char\_2):

return char\_1 + char\_2

reduce(join\_characters, list\_of\_characters, ‘**zz**’)

|  |  |  |  |
| --- | --- | --- | --- |
| Iteration | char\_1 | char\_2 | return (result of reduction step) |
| 1 | **zz** | **b** | **zzb** |
| 2 | **zzb** | **c** | **zzbc** |
| 3 | **zzbc** | **d** | **zzbcd** |

The final dictionary will have the following structure:

{

‘2005’: {

‘taxon1’: count for taxon1,

‘taxon2’: count for taxon2,

‘taxon3’: count for taxon3,

‘taxon4’: count for taxon4,

…

},

‘2006’: {

‘taxon1’: count for taxon1,

‘taxon2’: count for taxon2,

‘taxon3’: count for taxon3,

‘taxon4’: count for taxon4,

…

},

...

}

**Question 2:**

While working with data problems, you will frequently encounter ‘many-to-many’ mappings between various domain entities involved. A couple of examples:

1. “a gene may be associated with **many** diseases”; “a disease may be associated with **many** genes”.
2. in an online business, users and products also share this relationship - “a user may like **many** products”; “a product may be liked by **many** users”.

One way to represent this relationship is using nested dictionaries.

For ex., for different users and items in our online business, pairwise user-item ratings may be represented as

{ ‘user1’: {‘item1’: 1, ‘item2’: 3, …},

‘user2’: {‘item1’: 2, ‘item3’: 4, …},

‘user3’: {‘item1’: 4, ‘item4’: 5, …},

…

}

So, in this dictionary, user1 gives a rating of 1 to item 1, a rating of 3 to item2 etc. Note that not all users will have ratings for all items in the catalogue.

In this question, you’re given one such user-item-rating dictionary as above. The top-level **keys** of this dictionary are individual *users*. The **value** for each *user* is another dictionary (item-ratings) with individual *items* as keys and their ratings as *values*. Your goal is to perform the following steps:

**Step 1:** Write a function to reverse (transform) the user-item-rating dictionary to an item-user-rating dictionary, i.e. a dictionary where the top-level **keys** are individual *items*, and the **value** for each item is the corresponding user-rating dictionary. So for the above example, this function would generate:

{ ‘item1’: {‘user1’: 1, ‘user2’: 2, ‘user3’: 4, …},

  ‘item2’: {‘user1’: 3, …},

  ‘item3’: {‘user2’: 4, …},

  ‘item4’: {‘user3’: 5, …}

  ….

}

**Step 2:** Write a function that can take as input two ratings’ dictionaries, and compute/return some measure of their dis-similarity. Using the above example, for instance, the function may take as inputs item-rating dictionaries for two individual users from the user-item-rating matrix (where top-level keys are users), and generate user dis-similarity. The idea is that if two users rate the same items (for ex. movies) similarly, then they must have similar tastes as exhibited by their ratings. For now, compute the dis-similarity as: **Average of Absolute Difference of ratings for common keys**. For ex. with inputs {‘item1’: 3, ‘item2’: 4, ‘item3’: 5} and {‘item1’: 4, ‘item3’: 2, ‘item4’: 1}, the following assessments and computations are required:

**Common keys:** ‘item1’ and ‘item3’

**Difference in ratings for common keys:**

**Key ‘item1’:** (3 – 4) = -1

**Key ‘item3’:** (5 – 2) = 3

**Absolute difference in ratings for common keys:**

**Key ‘item1’:** abs(-1.0) = 1 ## use **import math** and use **math.abs(<x>)** method

**Key ‘item3’:** abs(3) = 3

**Average of absolute difference in ratings for common keys:**

(1 + 3)/2 = 2.0

**Note:** These sorts of operations are commonly used for making item recommendations to users using a technique called collaborative filtering. The (dis-)similarity function used above is an arbitrary heuristic, and may not work well in practice. But one can envision more complex pairwise scoring functions that work well with real datasets. And they exist.

**Step 3:** Use the above function to compute the dis-similarity of each pair of users in the user-item-rating dictionary, using their item-ratings. Likewise, compute the dis-similarity of each pair of items in the item-user-rating dictionary, using their user-ratings. Print the most similar (least dissimilar) user-pair(s) and the most similar (least dissimilar) item-pair(s) in the data provided.