Recap some of the File tricks

```
# Open a file to read

txt_f = open('readme.txt', 'r')
msg = txt_f.read()
txt_f.close()
print(msg)


# File handle is iterable

txt_f = open('readme.txt', 'r')
for line in txt_f:
   print(line)
txt_f.close()


# Open a file to write

txt_f = open('new.txt', 'w')
txt_f.write('Today is a good day.')
txt_f.write('It is a bless.')
txt_f.close()


# open a file to append at the end

txt_f = open('new.txt', 'a')
txt_f.write('Today is a good day.')
txt_f.write('It is a bless.')
txt_f.close()


# open a file for read and write

txt_f = open('new.txt', 'r+')
txt_f.write('Today is a good day.')
txt_f.write('It is a bless.')
txt_f.close()


# with statement

with open('info.txt') as f:
    data = f.read()
print(data)
```

```
============================================================
```

Use `if __name__ == '__main__':` for better code organization and allowing smooth autograding.

P1. Write a program to read `README.txt` file and print out its content.

Example: when run, we will see

```
==================================================
Antique yet up-to-the-minute,
familiar yet unrecognisable,
outwardly urban but quintessentially rural,
conservative yet path-breaking,
space-age but old-fashioned,
China is a land of mesmerising contradictions.
==================================================
```

See content of `README.txt`.

P2. Write a function named `simple_write`. The function takes 2 arguments: a filename and a string. The function is to write the string into a file of the given filename.

Example:

When run with the following code:

```
==================================================
s = "Yet the truely unique feature of our language " + \
"is not its ability to transmit information " + \
"about men and lions. Rather, it's the ability to " + \
"transmit information about things that do not exist at all." + \
"\n-- Yuval Noah Harari, Sapiens."

simple_write('new.txt', s)
```

```
================================================
```

Check content of `new.txt`.

It affects the `new.txt` file: if the file exists, the file is overwritten with the Harari's quote. Otherwise, it is created with Harari's quote as its content.

P3. Taxonomy nomenclature is a naming system biologists adopted for formally classifying living organisms. Write a function named `read_taxo`. The function takes a filename of taxonomic names, reads its content containing one taxonomic name per line and returns a list of taxonomic names.

Example:

When run with the following code:

```
================================================
res = read_taxo('whales.txt')
print(res)
================================================
```

we will see:

```
================================================
['Balaena mysticetus', 'Eubalaena glacialis', 'Eubalaena japonica', 'Eubalaena australis', 'Balaenoptera musculus']
================================================
```

See content of `whales.txt`.

P4. Write a function named `read_vec`. The function takes a filename, reads its content containing lines each has a number and returns a list of integers corresponding to the file content.

Hint: recall that `int('345')` *gives* `345`.

Example:

When run with the following code:

```
================================================
v = read_vec('vec.txt')
print(v)
================================================
```

we will see:

```
================================================
[1, 5, 9, 30]
================================================
```

See content of `vec.txt`. Notice that numbers are integer, not strings!

P5. Write a function named `read_records`. The function takes a filename, reads its content containing lines each has a pair of key and value separated by "=" sign and returns a dictionary of key-value pairs corresponding to the file content.

Example:

When run with the following code:

```
================================================
res = read_records('top_NP.txt')
print(res)
================================================
```

we will see:

```
================================================
{'Cambodia': 'Bokor', 'Malaysia': 'Kinabalu', 'Vietnam': 'Phong Nha-Ke Bang', 'Thailand': 'Khao Sok'}
================================================
```

Note: order may be different due to nature of dictionary. See content of `top_NP.txt`.

P6. Write a function named `write_records`. The function takes 2 arguments: a filename and a dictionary whose keys and values all are strings. Then, the function writes all key-value pairs to a file of the given name where each line contains a pair separating a key and its value by "=" sign. The key-value pairs are written in an ascending order using `sorted_keys` function from `checkdict` module provided with the template.

Use the P6 template and the auxiliary module. (`write_dict_template.py` and `checkdict.py`; note: template and auxiliary files are only to encourage intended learning skills and/or allow smooth auto-grading.)

Hint: remember string concatenation and `'\n'` *is a new line symbol.*

Example:

When invoked with the following code:

```
=====================================================================
world_heritage = {'Ayutthaya': 'Cultural 710 acre',
           'Sukhothai': 'Cultural 29,290 acre',
           'Thungyai-Huai Kha Khaeng': 'Natural 1,537,000 acre',
           'Ban Chiang': 'Cultural 160 acre',
           'Dong Phayayen-Khao Yai': 'Natural 1,521,000 acre'}

write_records('unesco.txt', world_heritage)
=====================================================================
```

we will see the content of the `unesco.txt` file as

```
Ayutthaya=Cultural 710 acre
Ban Chiang=Cultural 160 acre
Dong Phayayen-Khao Yai=Natural 1,521,000 acre
Sukhothai=Cultural 29,290 acre
Thungyai-Huai Kha Khaeng=Natural 1,537,000 acre
```

Note that since the example code does not have any print statement, we will not see anything interesting on the console. We need to **OPEN THE FILE** to see its content.

P7. Write a function named `read_mat`. The function takes a filename, reads its content containing lines of numbers separated by spaces and returns a 2D list of integers corresponding to the file content.

Hint: this problem is quite challenging. It requires multiple sub-tasks. Work gradually may help ease the task.

(1) We can easily create a new list by an empty list assignment, e.g., `row = []`. *Try*

```
mat = []   # create a new list and this new list is empty.
for i in range(3):
    row = [] # create a new list and this list is empty.
    for j in range(4):
        row.append(j)  # add j into list row for 4 times: j = 0,..., 3
    mat.append(row) # add row into list mat for 3 times: i = 0,...,2
print(mat)
```

What do we learn here? How many times `row = []` *is run?*

(2) We read strings from a file, but for each number we need to convert it to an integer, using int(…).

Use the P7 template. (`mat_template.py`; note: a template file is only to allow smooth auto-grading.)

Example:

When run with the following code:

```
================================================
res = read_mat('mat0.txt')
print(res)
================================================
```

we will see:

```
================================================
[[3, 4, 5], [8, 10, -6], [9, 5, 4], [77, 100, 3000]]
================================================
```

P8. Our gene information contains in every of our cells. Genes are coded with DNA. With DNA transcription and translation processes, our genetic code is used to guide the protein synthesis. Proteins are macromolecules serving various functions within organisms, including being parts of cells, catalyzing metabolisms as enzymes, being reactants in the metabolism, etc. In short, protein is a sequence of amino acids. Different sequences of amino acids define different types of proteins. DNA is a molecule composed of two strands of complementary pairs of nucleotide bases. There are 4 types of nucleotide bases, i.e., cytosine (C), guanine (G), adenine (A), or thymine (T). A sequence of the bases carries genetic information for protein synthesis, such that 3 bases are translated to 1 of 20 amino acids composing to be a protein (or a stop signal, indicating the end of sequence). A 3-base set is called a codon. Therefore, "given a codon table", protein---a sequence of amino acids--- can be synthesized from genetic DNA code. Note: codons are read in succession. They do not overlap each other. (See https://www.nature.com/scitable/definition/codon-155

for details)

Write a function named codon. The function takes 2 arguments: a file name of a codon table (as string) and a file name of a DNA sequence, reads the two files, uses a codon table to decipher the DNA sequence to a protein, and returns a list of the deciphered sequence of amino acids. The codon table file is a simple text file, written in a simple format: each line contains one codon and its corresponding amino acid with "=" in between. The gene sequence is a text file with the first line being a source URL and gene sequence is written from the second line on. Each line contains no more than 70 bases. (Note: fun gene data can be downloaded from NCBI GenBank, looking for download "FASTA")

Hint: (1) something like `read_records` *from P5 may be handy for codon table.*
(2) codon is a non-overlapping string and it goes by 3 bases, i.e., `range(0, N, 3)` *may be handy.*

Example:

When run with the following code:

```
==================================================================
```

```
res = codon('codons.txt', 'homo_sapiens_mitochondrion.txt')
print(res)
print(len(res))
```

```
======================================================================
```

we will see:

```
======================================================================
['Lysine', 'Glycine', 'Leucine', 'Alanine', 'stop', 'Leucine', 'Lysine',
'Tryptophan',   'Leucine',   'Isoleucine',   'Cysteine',   'Valine',
'Glutamine', 'Leucine', 'Methionine', 'Glutamine', 'Serine', 'Glycine',
'Valine', 'Leucine', 'Glutamine', 'Serine', 'Leucine']
23
======================================================================
```

See `codons.txt` and `homo_sapiens_mitochondrion.txt` for their content c.f. the results shown. Note: also test your code with `homo_sapiens_insulin.txt`, which is longer and has multiple lines.

Use the P8 template. (`codon_template.py`; note: a template file is only to allow smooth auto-grading.)

P9. Sweetness is often considered to be the most desirable taste. Sweetness is measured as perception comparable to sucrose (table sugar). For example, sucrose has sweetness of 1, fructose (found in fruits) is sweeter than sucrose has sweetness of 1.7. This is measured by perception test where tasters taste the solution compared to sucrose solution and a 10% fructose solution (10g of fructose filled up with water to 100mL) is on average perceived as sweet as a 17% sucrose solution (17g of sucrose filled up with water to 100mL). Soft drink is made up to sweetness of 10% sucrose solution. Getting to around 15% sucrose solution makes the drink tasted more like syrup.

Write a function named `sweet` to take 2 filenames: one is for a sweetness table and another one is for sweeteners in a drink. The function calculates estimated sweetness of the drink comparable to 10% sucrose solution, and appends its calculation to the end of the second file---the drink-sweetener

file. The sweetness file has its first line as a header. Then from the second line on, each line contains a substance name, its sweetness level, and its calories (we don't use calories here). The drink-sweetener file does not have a header. Each line in the drink-sweetener file contains a substance name and its weight (in gram). All substance weights are in solution of 100mL. Note: the calculation is similar to weighted sum. For example, given a cup of 100mL drink contains 5g sucrose and 5g fructose. For sucrose sweetness of 1 and fructose of 1.7, this solution is approximately equivalent to 5·1 + 5·1.7 = 13.5. That is, this drink is as sweet as 13.5% sucrose solution and the word "Sweet as 13.5% sucrose solution" should be appended to the end of the drink-sweetener file.

Example

Given `sweetness1.txt`

```
substance sweetness calories/gram
lactose 0.16 4
glucose 0.7 4
sucrose 1 4
fructose 1.7 4
saccharin 300 0
```

and `CocaPanda.txt`

```
glucose 3
sucrose 3
fructose 3
```

When run with the code:

```
================================================
sweet('sweetness1.txt', 'CocaPanda.txt')
================================================
```

The `CocaPanda.txt` will be updated to

```
glucose 3
sucrose 3
fructose 3
Sweet as 10.2% sucrose solution
```

Use the P9 template. (`sweet_template.py`; note: a template file is only to allow smooth auto-grading.)

P10. "Fire is the rapid oxidation of a material in the exothermic chemical process of combustion, releasing heat, light, and various reaction products." from Wikipedia: Fire. Combustion is a chemical reaction between fuel and an oxidant, like oxygen. It takes an activation energy to break the covalence bonds of the reactants and then after the products are formed, new covalence bonds are made and energy are released. The entire process releases energy more than it took, therefore it is exothermic. For example, burning methane

$$CH_4 + 2\ O_2 \rightarrow CO_2 + 2\ H_2O$$

requires an activation energy to break 4 `C-H` bonds and 2 `O=O` bonds and it releases energy from forming 2 `C=O` bonds and 4 `O-H` bonds in the products. Given `C-H` bond energy of 416 kJ, `O=O` bond energy of 498 kJ, `C=O` bond energy of 803 kJ, and `O-H` bond energy of 467 kJ, the activation energy is $4 \times 416 + 2 \times 498 = 2660$ kJ and the energy released is $2 \times 803 + 4 \times 467 = 3474$ kJ, which gives the difference of -814kJ.

Write a function named `fire`. The function takes 2 filenames: one is for covalence bond energies and another is for the reaction specifying amounts of reactant and product bonds. The function reads both files for necessary information, then calculates activation energy, releasing energy, and the different energy and appends the result at the end of the second file---the reaction-bond file. The bond-energy file contains bonds and their corresponding energies: each line has bond symbol and its energy (in kJ/mol), separated by a space. The reaction file has the first

line describing the reaction (we can ignore it). The second line specifies numbers of reactant bonds (required to break) in format: a number of bonds, bond symbol, '+', a number of bonds, bond symbol, and so on. The '+' sign is to separate numbers and bond symbols into pairs. The third line specifies numbers of reactant bonds (required to form) in a similar format to the second line. Note that a number of bonds can be floating point, e.g., octane burning $C_8H_{18} + 12.5\ O_2 \rightarrow 8\ CO_2 + 9\ H_2O$.

Examples

Given `bond_energy.txt`

```
C-C 346
C=C 835
C-O 358
C=O 803
C-H 416
O-O 142
O=O 498
O-H 467
H-H 432
```

and `methane.txt`

```
CH4 + 2 O2 -> CO2 + 2 H2O
4 C-H + 2 O=O
2 C=O + 4 O-H
```

When run with the code:

```
================================================
fire('bond_energy.txt', 'methane.txt')
================================================
```

The `methane.txt` will be updated to

```
CH4 + 2 O2 -> CO2 + 2 H2O
```

```
4 C-H + 2 O=O

2 C=O + 4 O-H

Ea = 2,660.0 kJ, Er = 3,474.0 kJ, E = -814.0 kJ
```

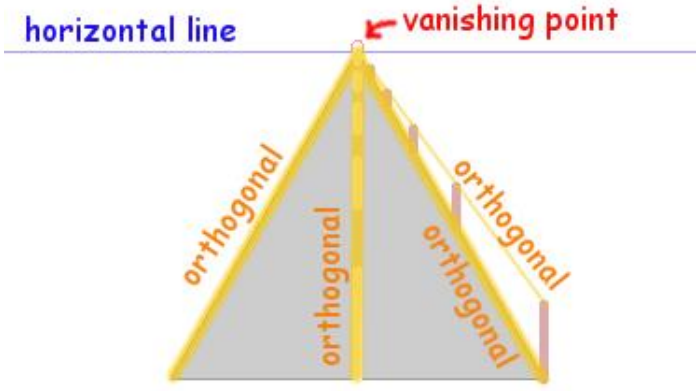
Use the P10 template. (`fire_template.py`; note: a template file is only to allow smooth auto-grading.) Try it with `octane.txt` as well.

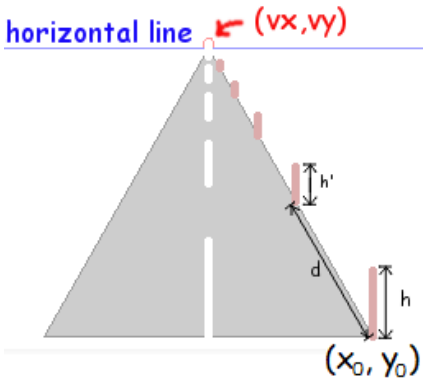
P11. Linear perspective is a visual technique to create an illusion of depth on a 2D screen. The technique is believed to be developed by renaissance architect and artist Filippo Brunelleschi. The technique has main 3 components: parallel lines (called orthogonals), the horizontal line, and a vanishing point. In short, as an object appears smaller as it locates further away, parallel lines, like a rail way, seem to converge at one point as they go further away from a viewer. That point to where lines seem to converge is called a vanishing point. Any parallel lines going toward a vanishing point are then drawn tilted toward the vanishing point. A length of any line perpendicular to the orthogonals is shorten accordingly. Note that the vanishing point is usually the point toward where artists intentionally lead viewer's eyes.

Write a function named `brunelleschi_lamp` to calculate a height of a lamp post on a road-lamp-post scene. The function takes vanishing point coordinates (v_x, v_y), a reference lamp post coordinates (x_0, y_0), height of the reference lamp post (h), a projected distance (d) between the reference lamp post and the lamp post under question, and a log filename, then calculates the height of the lamp post under question (h') according to linear perspective and return it out as well as **append** the calculation to a log file.

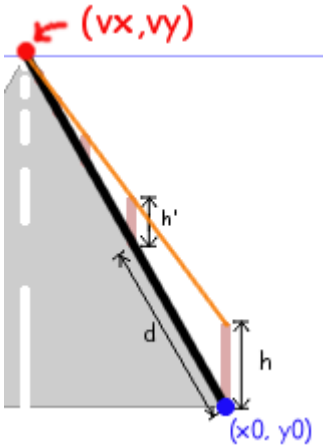
Use P11 template and an auxiliary file. (`lin_persp_template.py` and `mama_turtle.py`; note: template and auxiliary files are only to allow smooth auto-grading and better visualization.)

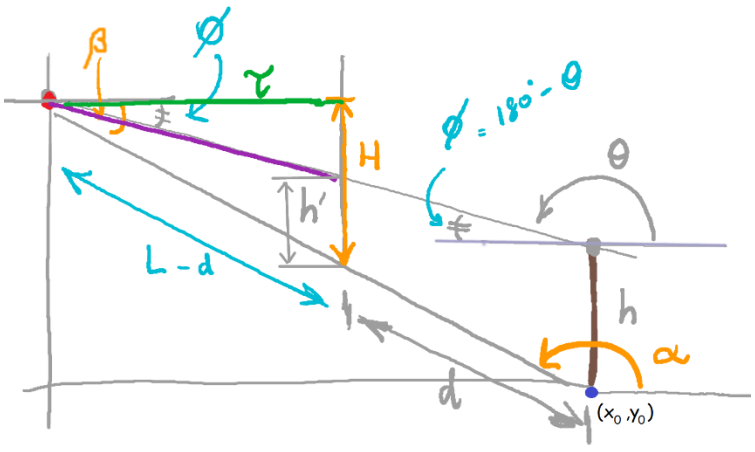
A road-lamp-post scene for this function is intended may look like the following illustrations.





When we consider (vx, vy), (x_0, y_0), d, and h, the height under question h′ can be deduced from geometry. The angle α pointing from reference (x_0, y_0) toward the vanishing point (vx, vy) and the length (L) between two point can be easily obtained: function `find_pol` of module `mama_turtle` takes care of this. Then, with the angle β (= π – α) and length L (actually L - d), H and τ can be deduced: $H = (L - d) \sin \beta$; $\tau = (L - d) \cos \beta$. Similarly, an angle θ pointing from top of the reference post (x_0, y_0+h) toward the vanishing point can be obtained (function `find_pol` again). Given H, τ, *and* angle ϕ = π - θ, h′ can be obtained from $(H - h')/\tau = \tan \phi$.





Example

When run with the following code

When run with the code:

==========

```
from mama_turtle import road
if __name__ == '__main__':
    r = road()
    r.draw_road()
    r.draw_lamp_posts(brunelleschi_lamp)
    input('enter to exit')
==================================================
```

Notice that `brunelleschi_lamp` function is passed to `draw_lamp_posts` method of `road` object imported from `mama_turtle` module (auxiliary file). The resulting `log.txt` (freshly created) may look like

```
Input: vx=0.00, vy=75.00, x=100.00, y=-100.00, height=40.00, d=0.00.
Calc: L = 201.56, beta = 1.05, H = 175.00, tau = 100.00, phi = 0.93, h' = 40.00.
Input: vx=0.00, vy=75.00, x=100.00, y=-100.00, height=40.00, d=94.06.
Calc: L = 201.56, beta = 1.05, H = 93.33, tau = 53.33, phi = 0.93, h' = 21.33.
Input: vx=0.00, vy=75.00, x=100.00, y=-100.00, height=40.00, d=141.09.
Calc: L = 201.56, beta = 1.05, H = 52.50, tau = 30.00, phi = 0.93, h' = 12.00.
Input: vx=0.00, vy=75.00, x=100.00, y=-100.00, height=40.00, d=169.31.
Calc: L = 201.56, beta = 1.05, H = 28.00, tau = 16.00, phi = 0.93, h' = 6.40.
Input: vx=0.00, vy=75.00, x=100.00, y=-100.00, height=40.00, d=188.12.
Calc: L = 201.56, beta = 1.05, H = 11.67, tau = 6.67, phi = 0.93, h' = 2.67.
```

This is due to that `draw_lamp_posts` method calls `brunelleschi_lamp` function for 5 times. (Recall: it is opening a file in an append mode.)

P12. (Super Special Problem! Caution: content may be too tough!) Travel is about learning other cultures, other places, other perspectives. Society and humanity are always at the center of this learning. To process this kind of non-structure data (i.e., it is a description, not a tabular data; some calls this natural language processing or NLP), various schemes have been invented to transform this non-structure-data message into some kind of structure data, preferably computable as well. Among those, TF-IDF is one of the most widely used. TF-IDF (term frequency-inverse

document frequency) is a measure of word importance in documents. It is designed to signify specific words uniquely appeared in some documents as well as tone down common words commonly appeared in all documents.

Write a function named `culture_tf_idf`. The function takes an argument: a list of filenames each describing a culture. The function goes through all files in the specific path, computes TD-IDF for every file and every word in it, and return a nested dictionary of the results. The resulting dictionary is nested in a way that the first level key is a document name and the second level key is a word. Note: use the following TF-IDF formulation for this problem:

$$tfidf = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \cdot log\left(\frac{N}{1+N_t}\right),$$

where *tfidf* is a TF-IDF measure of term *t* in document *d*; $f_{t,d}$ is a frequency term *t* appeared in document *d*; *N* is a number of all documents; and N_t is a number of documents having term *t*. The first half the of the product, $\frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$ is term frequency or *tf*. It is to quantify how often a term appears in the document. The second half of the product $log\left(\frac{N}{1+N_t}\right)$ is inverse document frequency or *idf,* which discounts the term if it is common in many document. For example, given 3 documents: A, B, C, where A has 'stupid is stupid does'; B has 'real strength is within' and C has 'wisdom is power', we will have

$f_{stupid, A} = 2$; $f_{is, A} = 1$, $f_{does, A} = 1$.

$f_{real, B} = 1$; $f_{strength, B} = 1$, $f_{is,B} = 1$, $f_{within, B} = 1$.

$f_{wisdom, C} = 1$, $f_{is,C} = 1$, $f_{power,C} = 1$.

N = 3 (for A, B, and C).

N_t: $N_{stupid} = 1$ (for A having stupid), $N_{is} = 3$ (for A, B, C all having it), $N_{does} = 1$, $N_{real} = 1$, ..., $N_{power} = 1$.

Therefore, $\text{tf-idf}_{stupid,A} = 2/(2 + 1 + 1) \cdot \log(\ 3/(1 + 1)\ ) = 0.2027$;

$$\text{tf-idf}_{\text{is,A}} = 1/4 \cdot \log(\ 3/(1 + 3)\ ) = -0.0719;$$

... $$\text{tf-idf}_{\text{power,C}} = 1/3 \cdot \log(\ 3/(1 + 1)\ ) = 0.1352$$

Hint: (1) find $f_{t,d}$ first (using a nested dictionaries for $f_{t,d}$ could make things easier); (2) find N and N_t (using a dictionary with keys being words for N_t could ease the task); (3) find $\sum_{t' \in d} f_{t',d}$ before calculating TF. This is a SUPER DIFFICULT PROBLEM.

Use the P12 template. (`you_must_be_crazy_template.py`; note: a template file is only to allow smooth auto-grading.)

Example

Given `chinese.txt`

```
China
Chinese speaks loudly.
They associate loudly speaking with good health.
They value good health and longevity highly.
```

`thai.txt`

```
Thai
Thai speaks softly.
They associate soft speaking with good manner.
They value fun and good food.
```

`japanese.txt`

```
Japan
Japanese is polite.
They speaks formally.
They value responsibility and politeness.
```

When run with the code:

```
======================================================================
```

```
def nice_print2Ddict(d):
    dkeys = list(d.keys())
    dkeys.sort()
    for k in dkeys:
        print('\n', k)
        d2 = d[k]
        k2s = list(d2.keys())
        k2s.sort()
        print('*', end=' ')
        for k2 in k2s:
            print("{}:{:.3f}".format(k2, d2[k2]), end='; ')


if __name__ == '__main__':
    cultures = ['chinese.txt', 'thai.txt', 'japanese.txt']
    res = culture_tf_idf(cultures)
    nice_print2Ddict(res)
```

==========

we will see something like

==========

```
chinese.txt
* and:-0.017; associate:0.000; chinese:0.024; good:0.000;
health:0.048; highly:0.024; longevity:0.024; loudly:0.048;
speaking:0.000; speaks:-0.017; they:-0.034; value:-0.017; with:0.000;
japanese.txt
* and:-0.026; formally:0.037; is:0.037; japanese:0.037; polite:0.037;
politeness:0.037; responsibility:0.037; speaks:-0.026; they:-0.052;
value:-0.026;
thai.txt
* and:-0.018; associate:0.000; food:0.025; fun:0.025; good:0.000;
manner:0.025; soft:0.025; softly:0.025; speaking:0.000; speaks:-0.018;
thai:0.025; they:-0.036; value:-0.018; with:0.000;
```

==========