0 Chronicles of the Python: The File, the With and the Try

The following code will create a file named $new.txt$, with a single line in it.

```
f = open("new.txt", 'w')
f.write("Hello, World!")
f.close()
```

The first argument of $open()$ is the filename, and the second argument of $open()$ is open mode, with the following options(not all of them are listed).

'r'	Only read the file (When no options are given, this is the default mode)
'w'	Newly write on the file (File with the same name will be deleted)
'a'	Append at the end of the file
'r+'	Read & Write Mode (Error if file does not exist)
'w+'	Read & Write Mode (Create a new file if file does not exist)
'rb', 'wb', ...	Open binary(non-text) files

After we are done with the file, we have to explicitly close it with $f.close()$, so that other processes can use the file.

Write a program that would print the contents of a file named $log.txt$. After printing, it would add a new line containing the current time to the same file.

1. Use the $with$ statement so that $f.close()$ can be automatically done.
2. Add appropriate error handling, with $try - except$, if necessary. Depends on which mode you choose to operate.

hint: You can get the current time by

```
import time
time.strftime('%H : %M')
```

Try out other options (%Y, %m, %d ...) as well!

1 Heart Signal

It is not often the case that someone you love, loves you back. It was easily observed in the heartwarming(or breaking?) show Heart Signal. Now, as the #1 fan of the show, it is our job to see if the participants can find the perfect match. Perfect match means that the participants like each other.

INPUT: The first line is integer N. The next N lines contain the name of all the participants, followed by the name of the person he/she likes, seperated by a space. The last line is the name of a participant.

OUTPUT: If the participant is also liked by the someone he/she likes, print **LOVE**. Print **OUCH** elsewise. Print **WHO?** if the name did not appear in the participant list.

EXAMPLE:

```
5
hulk blackwidow
hawkeye blackwidow
blackwidow hawkeye
ironman pepper
pepper ironman
hulk

OUCH
```

```
5
hulk blackwidow
hawkeye blackwidow
blackwidow hawkeye
ironman pepper
pepper ironman
me

WHO?
```

2 Statistics Showdown

Given $N(N\%2 == 1)$ numbers, compute the following representative values.

1. Arithmetic Mean
2. Geometric Mean
3. Median
4. Mode (The most common value)
5. Range ($max(n_i) - min(n_i)$)

INPUT: The first line is integer N. The *i*-th line of next N lines contains a single integer n_i.

OUTPUT: Print 5 lines, each consisting of the Arithmetic Mean, Geometric Mean, Median, Mode, and Range of n_i. If there are more than 1 mode, print any of those. For Arithmetic and Geometric mean, round to one decimal place (*e.g.* $1.75 = 1.8, 3.74 = 3.7$).

CONDITION: $1 \leq N \leq 200$ (N is odd number), $0 \leq n_i \leq 1000$

EXAMPLE:

```
5
1
4
2
3
5
```

```
3.0
2.6
3
1
4
```

HINT: Googling "root in python (other than square root)" would help calculating the Geometric Mean.

3 DIE with a T

I am on a diet. I can only eat upto 2,000 kcal (2,000,000 cal) a day. I eat 4 meals a day(*b*reakfast, *l*unch, *s*nack, and *d*inner) and life would be easy if every meal was exactly 500,000cal. But it isn't, so help me out. From 4 different groups of menus(each group representing the 4 meals I eat) each consisting of N foods, compute whether it is possible to select a menu from meal each group so that the total calorie intake is *exactly* 2,000,000 cal.

INPUT: The first line is integer N. The *i*-th line of next N lines contains the four calorie value of a menu in each meal group. The first columns are breakfast menus, the second lunch, and so on.

OUTPUT: Print **SUCCESS** if we can choose exactly 4 food so that the total of calorie equals 2,000,000. ($b_i + l_j + s_k + d_l = 2,000,000$) Print **FAIL** elsewise.

CONDITION: $4 \leq N \leq 10,000$, $0 \leq b, l, s, d \leq 1,000,000$

EXAMPLE:

```
2
100000 600000 400000 600000
200000 500000 600000 800000
```

```
SUCCESS
```

We can choose (200000, 600000, 600000, 600000) to eat $2,000,000$cal.

```
1
100000 600000 400000 600000
```

```
FAIL
```

hint: The obvious approach would be trying every combination of 4 foods from each meal group. However, this approach is $O(N^4)$, taking time proportional to 10000000000000000 in the worst case. Therefore, instead of trying every combination, we need a better approach. One possible approach would be computing *partial* calorie k in $O(N^2)$, and checking whether $(2000000 - k)$ is possible in $O(1)$. $Constant$ time membership check can be done by a certain data structure, which has the added benefit of removing duplicates, which will reduce computation time somewhat.

4 Optional: Parallelization

Download $mat\_mul\_skeleton.py$ to begin.
We will use parallelization to speed up matrix multiplication, which is a very common operation in modern computer programming. Of course, we will use the simplest form of multiplication & parallelization for convienience.

The key of parallelization is splitting the task into smaller, independent tasks. Matrix multiplication can be split by multiplying the lefthand matrix with each column of the righthand matrix:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} * \begin{bmatrix}5 & 6 & 7\\8 & 9 & 10\end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} * \begin{bmatrix}5\\8\end{bmatrix} \;\&\; \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} * \begin{bmatrix}6\\9\end{bmatrix} \;\&\; \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} * \begin{bmatrix}7\\10\end{bmatrix}$$

We will use *Pool().map(f, ll)* to parallelize the multiplication. f is a function that multiplies the lefthand matrix to a single vector, and ll is a $list$ of $list$ where each item($list$) is a column of righthand matrix.

Other smarter representation of matrices like $numpy.array()$ provide column-wise access. However, since we use $list$ of $list$ to represent matrices column-wise access is frustrating. Therefore, we will transpose the matrix so that each column now becomes the row. $transposed(matrix)[0]$ would return the first column of the original matrix. We then apply f to each row of the transposed matrix, then transpose it back.

I have implemented the $transpose$ and single-core matrix multiplication. All we have to do is fill in the $\#TODO$, and try experiments, varying the size of matrices. Which one is faster when matrix sizes are small? when large?