General Notes for File I/O Assignments

All solution functions that produce files will produce files that have '_soln' appended to the file. For example, if your function is supposed to produce the file 'example.txt', the solution file will produce 'example_soln.txt'.

You should also compare your files to the solution files using MATLAB's file comparison tool: http://www.mathworks.com/help/matlab/matlab_env/comparing-files-and-folders.html

Some things to watch out for when creating files are making sure that you do not have extra newline characters at the end of your file and that your spaces are actually ASCII value 32. The file comparison tool will catch both of these errors, but it is still a good thing to be mindful of beforehand.

Function Name: `detention`

Inputs:

1. (*char*) A string with a sentence to be copied
2. (*double*) The number of lines to print
3. (*double*) The number of times per line the sentence should appear

Outputs:

(none)

File Outputs:

1. A text file of lines written in detention

Function Description:

Throwback to old-school detention where you are told to copy down a sentence over and over. However, that's way too tedious so you decide to use MATLAB instead!

Write a function that takes in an input of a sentence along with the number of lines to write and the number of times per line the sentence should appear.. Then, write a file with all of the copied sentences. Each line should begin with `'Line <num>. '` and each sentence should have a space between it and the next sentence. The output file name should be the string you are supposed to copy, but converted to camelcase (feel free to use your `camelCase` function from Homework 5 for this).

For example, if the input string is `'MATLAB is the best!'` and it was to be written 4 times with 2 sentences per line, the output file should be named `matlabIsTheBest.txt` and should look like this:

```
Line 1. MATLAB is the best! MATLAB is the best!
Line 2. MATLAB is the best! MATLAB is the best!
```

Notes:

- There should be no extra spaces at the end of each line
- There should be no extra blank lines in your file (this would occur if your last line has a new line character at the end of it).

Function Name: `hashtagMATLAB`

Inputs:

1. (*char*) A string representing the name of a .txt file that contains tweets

Outputs:

1. (*char*) A string of the username and number of hashtags used

Function Description:

The rise in social media over the years has been accompanied by the development of a new found vocabulary. Acronyms and words that once either had no meaning or a completely different use are now being used every day; one such word, of course, is 'hashtag'. In case you have not fallen victim to these social media terms, a hashtag is "a word or phrase preceded by a hash or pound sign (#) and used to identify messages on a specific topic" (Google). It is commonly used in the popular social networking service Twitter.

You have recently obtained an internship at Twitter and are eager to get started. Your first task is to go through every user and determine how many hashtags they've used. Having taken CS 1371, you decide to let MATLAB do all the heavy lifting. Luckily, Twitter has supplied text files of their users' tweets to make your life easier. Write a function that will look through a text file of a user's tweets, find the username and count the number of hashtags used. Once this information is found, you should output a string in the following format:

```
'<username> has used <number of hashtags> hashtags.'
```

For example, given the text file `'JohnMatlab.txt'` that contains

```
John Matlab
@MATLAB_Slayer412

Hitchhiker's Guide to the Galaxy got it wrong, the meaning of life is
strtok.
#realLife #MATLAB

When in doubt, iterate it out.
#trueAdvice #MATLAB
```

The output should be:

```
'MATLAB_Slayer412 has used 4 hashtags.'
```

If the user has only used 1 hashtag, then the output should read:

```
'<username> has used 1 hashtag.'
```

Notes:

- The filename will be the upper camelcase of the user's real name; this is the same as camelcase, but the first letter is also capitalized.
- The username will always be be preceded by an `'@'` and will be the line after the user’s real name. `'@<username>'` will be the only text in the given line.
- The name and username may not always be in the first and second line.
- It is possible for there to be more than one `'@'` in the file.
- You are guaranteed to find at least one hashtag.

Function Name: `deAdamize`

Inputs:

1. *(char)* the name of a text file containing puns
2. *(char)* the name of a text file with pun replacements

Outputs:

(none)

File Outputs:

1. A text file of the edited version of the original text

Function Description:

As you will come to find out throughout this semester, some TAs lack the ability to make quality jokes; one TA in particular has been honored in the name of this function. You have decided to take it upon yourself to remove the lowest form of joke, the pun, from various text files.

This TA only has a set amount of jokes he knows so the pun keywords you will always be looking for are `'punny'`, `'a salted'`, `'appeeling'`, `'ribbiting'`, and `'taco'`. However, after you find these puns you must go in and replace them with an appropriate substitute defined in the text file.

You are given a text file of appropriate replacements for each pun word. Each line of the text file will be formatted as:

```
<horrible pun>:<replacement>
```

Swap all pun words in the original file with their replacement and write the result to a new file. The new filename should have `'_edited'` appended to the original file name. For example: if the original filename was `'speech.txt'` the output file would be `'speech_edited.txt'`.

The puns will only be the ones listed above, but the replacements for these words could change.

Notes:

- Neither the pun nor the replacement will contain a colon (`:`).
- There could be multiple puns on one line.

Function Name: `bandName`

Inputs:

1. *(char)* The name of a .txt file containing information on how to make a band name

Outputs:

1. (*char*) The name of the band retrieved from the given files

Function Description:

You and your friends decided it was time to form a band with all of the musical talent you have. The only problem now is choosing a name. Since you and your friends can't all agree on a name, you've decided to write a function in MATLAB to help you pick out a random name! You will take in a file with three comma separated integers on each line. The first number identifies a lyric file. The second number identifies a line in that file and the third number identifies a word on that line. For example, if the line is `'3, 5, 8'`, you should open `'lyrics3.txt'` and add the 8th word of the 5th line to the band name.

Once you have gone through all of the lines in the file, you will have your band name chosen! Your output should be this name, with each word separated by a space. You should remove all punctuation from the band name except for apostrophes.

Notes:

- You will not be provided any file, line, or word number that does not exist.
- You do not have to worry about capitalization; whatever the capitalization is from the file is what should be used in the output.

Function Name: `bracketCheck`

Inputs:

1. *(char)* The name of a text file

Outputs:

1. *(char)* A string identifying balanced or unbalanced brackets

Function Description

The MATLAB Editor does many things to help you code such as indent lines and underline syntax errors. Another thing it does is highlight when there are unbalanced parentheses or brackets. This function will perform such a check on any text file. The input will be the name of a text file and the output will be one of two strings. If all curly braces, `{}`, square brackets, `[]`, and parenthesis, `()`, are balanced in the text file, then the output should be:

`'All brackets are balanced.'`

Otherwise the output should be:

`'The brackets on line <num> are not balanced.'`

Where `<num>` is replaced with the **first** line number where a mismatched bracket, brace or parenthesis occurred.

An unbalanced bracket is defined as any case where an opening bracket does not have a corresponding closing bracket and every closing bracket has a corresponding opening bracket. For example, '`{ here }([{ is } some ( text )])`' is balanced because every opening bracket is paired with a closing bracket that appears in the reverse order of the opening brackets and vica versa. `'{this(is) not balanced]'` is not balanced because the opening curly brace does not have a corresponding closing brace and the closing square bracket does not have a corresponding opening bracket. There may be any amount of other text between any of the brackets. You can assume that all brackets should be closed on the same line (if they are not, then you should consider the brackets to be unbalanced; you do not have to check if they are closed on a subsequent line).

Notes:

- The input text file will always contain at least one line.
- Be sure to check different cases against the solution file.