(I) Architectural Overview

(A) Data Structures

(1) Word Counts: Dictionary mapping words to their respective counts in the input file
(2) Joined Words: Dictionary mapping individual words to the joined words they are in
(3) Counts To Words: Dictionary mapping word counts to lists of the words having those counts
(4) Counts: List of all the counts of each individual word in the input file that is kept sorted
(5) Map Scores: Dictionary mapping the score of the current set of extracted words to a tuple consisting of a dictionary of the extracted words to their respective counts, the number of high frequency words deleted per iteration, and the number of low frequency words deleted per iteration.

(B) Main Ideas

(B1) Processing the Input

(1) Have a way of "standardizing" the text from the input file. In the case of this program, this involves stripping out all non alphanumeric and space-related characters, and converting all individual words to be in lowercase before adding them into the actual dictionaries.
(2) Try to get the majority of the two word terms that end up in our Dictionary to be the Proper Nouns from the input file.
(3) There needs to be a way of making sure that there is no overlap between two word terms. The joined_words dictionary will provide us with a quick way of checking whether or not a particular term is already a part of a two word term by simply checking if a certain word is in the set of keys of the dictionary.
(4) I made the decision of counting all instances of the individual words in two-word terms by incrementing ONLY the corresponding two-word term(s). This is once again accomplished by utilizing the joined_words Dictionary in order to access the two word term that corresponds to the current word in question, and incrementing that two word term's entry in the word_counts Dictionary.
(5) I used the readlines() function in order to correctly process the entire input file and ignore any intermediate gaps caused by spaces.

(B2) Performing Term Extraction

(1) When I was testing tweaking the values for the number of high frequency and low frequency elements to eliminate at each iteration of the loop, I realized that there was a great amount of variability between the exact numbers that worked for different articles, so in order to provide a larger amount of customizability from article to article, I decided to let the

perform_term_extraction function take in two parameters in addition to the output of the process_input function.
(2) Because of the fact that the counts_to_words dictionary maps the counts of individual words to lists of the corresponding words with the specified count, I decided to make an additional helper function (remove_word_with_count) to help eliminate individual words.
(3) I sort out the list values with multiple words in counts_to_words in terms of word length (in ascending order), and then pop out the value at the 0th index of the appropriate list at each iteration of the remove_word_with_count function. The reasoning behind this decision is that (in general), longer words tend to be more important than the shorter words.

(B3) Selecting Optimal Set of Extracted Words

(1) The ratio of high frequency elements to low frequency elements to eliminate at each iteration of the extraction process varies widely from article to article, so I decided to call the perform_term_extraction function on a variety of combinations of high_term_deletions and low_term_deletions, then assign heuristical scores to each of the resulting sets of extracted words, and finally choose the top scoring set of extracted words from these for each of the articles.
(2) In order to provide more information, I decided to also print out the optimal combinations for high and low deletions per iteration for each of the different articles.

(II) Function Overviews

(1) Process Input: Function that takes in the name of the input file and outputs a tuple consisting of the following data structures:

- **(1a)** Word Counts Dictionary
- **(1b)** Counts to Words Dictionary
- **(1c)** Joined Words Dictionary
- **(1d)** Counts

(2) Perform Term Extraction: Function that takes in the tuple of 4 elements generated by Process Input function and keeps on deleting terms from the word_counts and counts_to_words dictionaries until there are between 3 and 7 terms remaining in total.

(3) Remove Word With Count: A helper function that is called multiple times from the Perform Term Extraction function in order to delete words with highest and lowest remaining frequencies. Called once per deletion.

(4) Print Map and Extracted Terms: A helper function that prints out the two main desired outputs: (a) the map of words to counts that results from iterating over the file, (b) the list of 3 to 7 terms that is ultimately returned by the Perform Term Extraction function call.

(III) Runtime Analysis

(1) Process Input ~ O(size of the input file (# characters))
(2) Perform Term Extraction ~ O(total number of unique words in input)
(3) Remove Word With Count ~ O(1)
(4) Print Map and Extracted Terms ~ O(size of input (# words) + size of input(# characters))

(IV) Inputs Given

(1) tax_bill.txt: Article about recent tax reforms.
(2) Moodys_India.txt: Article about India's economy.
(3) keystone_pipeline.txt: Article about developments on the Keystone Pipeline project.
(4) lonzo_ball.txt: Article about the recent match between the Lakers and 76ers.

The reason I have provided multiple inputs is to illustrate how the program calculates the "optimal" ratio for the number of deletions of high frequency words to the number of deletions of low frequency words per article that is given as input. Also to illustrate how it tackles vastly different articles.

(V) How to Run

Run python representative_words.py from Command Prompt or Terminal.

*** Adding More Test Cases**:
All test cases are currently at the bottom of the file representative_words.py file, and are written as individual calls to the print_map_and_extracted_terms function. More can be added here fairly easily.