**Overview**
The purpose of this program is to facilitate phrase query searching across a set of documents and evaluate its performance using metrics such as TF-IDF score and cosine similarity.

The raw data which consists of 249 text files is preprocessed using the following steps: converting all text to lowercase, tokenizing all words, and eliminating stopwords, punctuation symbols and empty space. The preprocessed text files are stored in the directory preprocessed_data. Each file in this directory is iterated through to create a positional index which stores each unique word along with the title of the document it appears in and its position in that file.

During this, two other dictionaries are populated, term_count to store the number of times each term appears in every doc, and term_total_max to store the total number of terms in each document and the term that appears the most amount of times to use for later in Q2. It's important to note that the numeric value that represents the word's position is the count of words, not chars. For example, the positional index for the word conceptions is as follows: conceptions: gulliver.txt [1069, 53271, 88210]; radar_ra.txt [4126], where conceptions is the 1069th word in gulliver.txt.

To facilitate phrase querying for phrases up to five words, a new Python dictionary is created to store each word in the phrase along with its associated documents and positions. The posting lists will also merge, for example: if the phrase is: "insomuch conceptions", the posting list in the new dictionary will look like: gulliver.txt [1021, 1069, 53271, 88210] and radar_ra.txt[4126], as the individual posting lists in the positional index look like: insomuch: gulliver.txt [1021], conceptions: gulliver.txt [1069, 53271, 88210]; radar_ra.txt [4126]. The position values are sorted to be in ascending order.

The final step is to check if there are any sequential position values which indicate these words occurring one after the other. It is imperative to make sure that the number of sequential words is equal to the number of words in the input phrase to make sure that the entire phrase does indeed occur in the document. For example, the phrase "above the clouds" merges the posting lists for "above", "the", and "clouds". Each document is then checked to see if there are three position values with a difference of one. In this case, the output is: hitch2.txt: [1040, 1041, 1042] which means that the phrase is found in this document.

For Q2, first a dictionary used to store each idf value per term is created by iterating through the positional index and taking the log of the total amount of documents divided by the length of each positional index term's entry (effectively counting all documents a term has appeared in) + 1.

A dataframe, for the TF-IDF matrix, populated by 0s on initialization is created, its index consists of the vocabulary and the column headings are each of the document names. The user is then asked which TF weighing variant of five they'd like to use to fill in its values. This affects which

TF weighting scheme is utilized while the program iterates through term_count. Term_total_max is used for the ‘Term Frequency’ and ‘Double Normalization’ formulas.
A query is asked to be inputted for the query vector. The query is then preprocessed and tokenized, and the amount of times each term appears in the query is counted. Query_vector is initialized as a series populated by 0s with an index of the vocabulary. For each token in the query, its respective value in query_vector is set to the amount of times it appears in the query. This assumes that the user will only enter terms which are found in the vocabulary.

Afterwards, two metrics for similarity are calculated, dot product and cosine similarity. The document columns of the TF-IDF are iterated through, and the scores are stored within rank and rank_cos respectively. The top five scores from both dot product and cosine similarity are then printed.

**Comparison of TF weight variants**

This will be an overview of the different ways in which each TF variant differs and their different advantages and disadvantages.

Binary is a very simple variant which requires no counting of each term within documents, and documents only whether or not the term is present. It is thus easier than other variants to calculate, however does a poorer job on determining relevance since frequency does not matter, a document which mentions each token in a query sparsely is weighed the same as a document in which each token is relevant to each other. A plus side to this is longer documents do not have a bias towards it as opposed to shorter documents.

Raw count improves upon binary in which it counts the amount of times each term is mentioned in a document. This can result in retrieving more documents with a greater relevance, however is biased towards longer documents.

Term Frequency divides the raw count of a term by the total amount of terms in the document. This gives the ratio of how much the tokens of a query may make up the document as a whole. Therefore, the bias for a much longer document with more mentions of a token but less of a ratio will be deemed less relevant to a shorter document with a larger token:total terms ratio.

Log normalization helps further differentiate relevances by dampening the effect of IDF through normalizing the TF scores logarithmically.

Double normalization, the raw count divided by the count of the most frequently occurring term, further helps to prevent biases towards longer documents.