Project 4

Please read the entire document carefully! I would suggest reading the entire document *before* writing any code.

1 Introduction

For this project, you will write a program to generate a *concordance*. A concordance is an alphabetized list of the words (usually the important ones) from a text, along with their locations in the text. Your implementation will use a hash table to accomplish this.

2 Implementation

2.1 Hash Function

For all of the following, you will be using strings as keys to your hash tables. This means that we need a function that takes as input a string, and returns an integer. You'll *must* use the following hash function (a slight modification of `DJBX33A`) as the hash function for your hash tables:

$$\text{DJBX33A}(str) = 5381 * 33^n + \sum_{i=0}^{n-1} \text{ord}(str[i]) * 33^{n-1-i} \qquad \text{where } n = \min(\text{len}(str), 8)$$

Note that this may be implemented more efficiently than a direct translation of the above mathematical expression; this however is not required.

2.2 Concordance

A typical method of constructing a concordance begins with splitting the input by sections (or lines, verses, chapters, etc), and then into tokens (words, essentially). Then, each token is considered, and non-important ones are discarded. This includes numbers and any "common" words that are not interesting enough to keep track of. For this project, a list of words to ignore is provided known as the *stop-list*. Important words (words that were not filtered) are entered into a hash table, along with what section (or line, verse, etc) on which they were found.

Once the input has been processed, the resulting set of words is output in alphabetical order, along with the (distinct) locations at which they appeared (also in order).

A hash table is perfect for both aspects of this task, storing the stop words and building the concordance itself. This will allow fast lookup of words that occur multiple times in the file.

For this project, you will be reporting the concordance of a single text file, tracking the lines that on which each word appeared. So, for example:

Sample stop-words file

```
a
about
be
by
can
do
i
in
is
it
of
on
the
this
to
was
```

Sample text file

```
This is a sample data (text) file to be processed
by your word-concordance program!

A real data file is much bigger.
```

Then the output should be:

```
bigger: 4
concordance: 2
data: 1 4
file: 1 4
much: 4
processed: 2
program: 2
real: 4
sample: 1
text: 1
word: 2
your: 2
```

Notes:

- Words are defined as sequences of characters delimited by any non-letter (white space, punctuation). Apostrophes will be ignored completely.
- There is no distinction made between upper and lowercase letters (e.g. CaT is the same word as cat). To accomplish this, we will make everything lowercase (by calling `str.lower()`).
- Blank lines are counted in line numbering.

So, the general algorithm for the program will be:

1. Read the stop words file into a hash table (we want $O(1)$ ability to check if a word is a stop word). Note that this will use your implementation of a hash table. Also note that the values aren't important, all we really care about are the keys.
2. The word-concordance will be in a separate hash table. Process the input file one line at a time to build the word-concordance. This hash table should only contain the non-stop words as the keys (you will use the stop words hash table to filter out stop words). Associated with each key is its value, where the value consists of the line numbers where the key appears. *Do not include duplicate line numbers.* This should be efficient (i.e. $O(1)$).
3. Generate a text file containing the concordance words printed out in alphabetical order along with their line numbers. You may use Python's builtin sorting here. There should be one word per line followed by a colon and spaces separating items on each line.

See the provided Python files for specifics of each function you are required to write.

For processing the lines of the file:

- I recommend processing the file one line at a time (e.g. I'd suggest using a `for` loop over the file).
- For each line in the input file:

  - Convert the line to lowercase.
  - Remove all occurrences of the apostrophe character (`'`)
  - Convert all characters in `string.punctuation` to spaces. (That is the variable `punctuation` in the builtin `string` module.)
  - Split the string into tokens using `str.split()` (with no parameters).
  - Each token that is only alphabetic (i.e. `str.isalpha()` returns `True`) is considered a "word". All other tokens should be ignored.

3 Testing

You are provided with several text files to be used for testing. You are also encouraged to write your own!

The text files are:

- a stop words file `stop_words.txt`
- six sample data files that can be used for preliminary testing of your programs:
  - `file1.txt`/`file1_sol.txt`: contains no punctuation
  - `file2.txt`/`file2_sol.txt`: contains punctuation to be removed
  - `declaration.txt`/`declaration_sol.txt`: a larger file for testing
- six large data files that can be used for performance testing. DO NOT include any of these in your submitted tests. They may take too long to run and will be killed by the grader. That said, none of these should take longer than a few seconds.
  - `dictionary_a-c.txt`/`dictionary_a-c_sol.txt`
  - `file_the.txt`/`file_the_sol.txt`: for this you should use the empty stop words file
  - `war_and_peace.txt`/`war_and_peace_sol.txt`

You are *required* to achieve 100% test coverage. This means that every line of code in your functions *must* be executed at some point in at least one of your tests.

As discussed in the syllabus, when I grade your code, you will ordinarily receive feedback regarding any tests that fail, *unless* you do not have 100% test coverage. In the event you do not have 100% test coverage, the only feedback you will receive is that you need to do more testing. I don't want you using my grading script to do your testing in the last day.

4 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 0, as needed, to remember how to push your local code.

The files you need to have submitted are:

- `hash_table.py`
  - Your correct implementation of the `HashTable` from Lab 8.
- `concordance.py`

  - Contains the functions for creating the concordance.

- `concordance_tests.py`
  - Contains tests for all your required concordance functions. These should run on *any* correct implementation of this project. As such, you should not include any tests for helper functions.

- `concordance_helper_tests.py`
  - Optional file containing tests for all your helper functions. These tests will be used in conjunction with your `concordance_tests.py` to check for 100% coverage.

- Any text files you use for testing.
  - If you use a text file for testing, you'll need to submit it so that your tests run on my end.