Task 1: clean.py

This week, we're asking you to complete only two tasks. Task 1 is worth one-third of the total possible points for the assignment (excluding style points), and Task 2 is worth two-thirds of the total possible points for the assignment.

A crucial (if somewhat hated) step of any form of data analysis is *data cleaning*. Most data you'll deal with in the real world will be inconsistent, messy, or formatted for a different purpose. Some of these flaws come from humans (think data entry errors) and others come from technical sources (think corrupted files). An analyst cleans data by creating little programs to remove the errors, resolve the inconsistencies, and recover the corrupted information. Data cleaning tends to be a painstaking process.

The other task in this problem set processes texts at the level of the individual word, as opposed to the line, sentence, paragraph, or character. To clean a text so it is ready for analysis, we must transform paragraphs of information into lists of individual words. *Your task is to write a program that will isolate the individual words of a text from all of the whitespace and punctuation in the input text file.*

Create a program called `clean.py`, which turns files containing arbitrary text into lists of cleaned words. These clean words will be written to a second file, one word per line (the first time all semester we've written data to your hard disk). These files full of cleaned words will be analyzed in the other task of this assignment. Remember that we will test your code using different text files, so you might try to download texts from the Internet to stress test your code (most public domain books are available in `.txt` form).

`clean.py` reads two filenames as input: the name of a file containing arbitrary text (such as a Supreme Court opinion) and the name of a file to which it should store the result of the cleaning process.

We have provided you with some starter code to help you get up and running. Specifically, our starter code already includes a few lines at the beginning that read the contents of the input text file into a string variable called `text` and a few lines at the end that write the content of a list of strings called `words` to the output text file (one word per line).

You will need to fill in the rest of the file, which asks for the file names as input, cleans the string stored in `text`, and stores the resulting strings into a list called `words`. Your cleaning process should perform the following steps in the following order:

1. Remove any of the following punctuation from any place in the text: `,!.?;"()\:-'`. We have provided these characters in a constant called `PUNCTUATION` at the beginning of the starter code. (Hints: (1) remember that strings are lists and (2) consider the string `replace` method.)

   Be sure you understand what we are asking you to do in this step. You are meant to *remove* punctuation, even if doing so results in something that is no longer an English word. For example, this will turn the contraction "isn't" into the non-word "isnt"; it will mush together two words surrounding a hyphen turning "writing-desk" into "writingdesk"; and it will even mush together two words surrounding a lot of spaceless punctuation turning "Happiness.--That" into "HappinessThat". You are not expected to "fix" these cases, and thus your end product will include a lot of "words" that aren't proper English words as a result.

2. Convert all words to lowercase. The end result should contain `wiretap` rather than `Wiretap`. You even want words that properly have an uppercase letter to be lowercased, such as `i`.

3. Break the text into a list of words, in the process removing all whitespace between words.

4. Remove any words that do not consist solely of letters. (e.g. `brandeis.txt` contains citations, and none of their volume numbers or page numbers should end up in the clean data.)

Take care to use exactly these two variable names, `text` for the input string and `words` for the list of cleaned words. You can use other, temporary variables to store related data in between, but start with `text` and end with `words`.

We include a few examples below. Note that the shell command `cat` prints the contents of a file to your shell.

```
$ python clean.py
Input file: alice.txt
Output file: alice_clean.txt
$ cat alice_clean.txt
...
found
herself
at
last
```

```
in
the
beautiful
garden
among
the
bright
flowerbeds
and
the
cool
fountains


$ python clean.py
Input file: brandeis.txt
Output file: brandeis_clean.txt
$ cat brandeis_clean.txt
...
against
that
pernicious
doctrine
this
court
should
resolutely
set
its
face
```

A word of warning: Since this is the first time you have created a program that can write to a file, we'll take a moment here to give you a word of warning. When Python writes to a file, it permanently deletes any pre-existing versions of the file. For emphasis, *there is no way to recover a file once you have overwritten it with Python.* If you use Python to write to a file called `very_important_brief.docx`, you will permanently delete the existing version of `very_important_brief.docx`. To keep you from shooting yourself in the foot, the starter code for `clean.py` will print an error message if you tell it to write its output to a file that already exists. This error will be annoying: you will have to manually delete the old version of your output file before you can run the program again with the same output filename. Please know that we are doing this for your own good. If you modify elements of the starter code that the comments say not to modify, you are on your own.

Task 2: spell.py

This week, we're asking you to complete only two tasks. Task 1 is worth one-third of the total possible points for the assignment (excluding style points), and Task 2 is worth two-thirds of the total possible points for the assignment.

Now, let's build a spell checker.

Part A: The Spell-Checker

Because it can get a bit ungainly working with large files with thousands of words, we've given you a small example that you should work with initially. The file is called `declaration.txt` and contains the first two paragraphs of the Declaration of Independence, 353 words long, with some errors intentionally introduced by us. It is meant to be used with `word_list_small.txt`, which contains only 296 words and is derived from the words of `declaration.txt`.

Once your code works well on `declaration.txt`, you can move on to the other two, longer texts, `alice.txt` and `brandeis.txt`, to which we have also intentionally introduced some errors. Both of these larger files are meant to be used with a more complete word list file called `word_list.txt`.

Remember that you're not supposed to be working with `declaration.txt`, `alice.txt`, or `brandeis.txt` directly in this task. Instead, request as input the name of a text file full of words as generated by `clean.py`. (e.g. `alice_clean.txt` or `brandeis_clean.txt` in the previous examples.) Also request as input the name of the appropriate word_list file. (The word_list files have already been cleaned just as if they had been run through `clean.py`.)

For each word found in the input text that is not in the word_list, your program should report a potential misspelling.

To emulate the spell checker you are used to in Microsoft Word or Google Docs, *your program must also recommend words from the word_list file that the user might have meant instead of the misspelled word, by applying a few intelligent heuristics.* To a computer programmer, a *heuristic* is a rule-of-thumb that is effective an acceptable amount of the time.

Your spell checker won't be quite as intelligent as commercial spell checker: it will explore a limited set of heuristics, searching only for mispellings that are "off by one". The heuristics you should test are:

1. Did the user user insert a letter that didn't belong in the word, so instead of spelling `bread`, they typed `breead` or `breatd`?
2. Did the user simply swap two consecutive letters, such as `braed` or `rbead` or `breda`?
3. Did the user mistype a single letter, replacing it with an incorrect one, such as `btead` or `breaf`?

Whenever you detect a misspelled word, apply each of these heuristics one-at-a-time to generate a long list of every possible string of characters the user might have meant to have typed instead. To be clear, most of the candidates you generate will be gibberish that aren't real words. We'll get rid of all of those gibberish words in a later step. For now, just come up with every word the user might have meant to have typed.

You must test these three heuristics exhaustively. So for the "swap" heuristic, number two, you need to analyze the word with the first two characters swapped, the second and third characters swapped, the third and fourth swapped, and so on all the way to the final two swapped. Likewise, for the first heuristic, you need to generate candidate words that drop every single letter from the word, one-at-a-time, from the first letter to the last.

Figuring out how to generate all of these subtle variations is probably the hardest part of the assignment. As a hint, to apply each heuristic, break up the misspelled word into smaller parts that you then reassemble through concatenation. So, let's say the input includes the misspelled word `breatd` rather than `bread`. To apply the first heuristic (the user inserted a letter), you will need to generate all of the following possibilities:

```
reatd
b + eatd
br + atd
bre + td
brea + d # This is the right one
breat
```

You'll need to use a for-loop to generate these six possibilities. Inside this for-loop, how do you create the small sub parts on each line above (such as `eatd`, `br`, or, `td`)? The answer is by generating slices with carefully selected indices. Pay close attention to our coverage of slices and concatenation in lecture and lab.

An additional hint: for the third heuristic, the user mistyped a letter, notice that we don't know which letter the user meant to type instead. To reconstruct candidate words (including many gibberish words), you need to try every letter from a to z in every spot in the word from beginning to end. To implement this, we recommend you place the following constant at the beginning of your file:

```
ALPHABET = 'abcdefghijklmnopqrstuvwxyz'
```

Then, you can use this variable as a way to iterate over every letter in the alphabet. (What loop can you construct to do this?)

Once you have created a list of every single possible correction produced by every heuristic listed above (it might be hundreds of words long), you must next get rid of the gibberish non-words. To do this, you must filter out those that do not appear in the word_list. The small number of words that remain are possible corrections that you should recommend to the user.

As output, whenever your program detects a misspelling, print on one line: the word, the characters `->` surrounded by spaces, and the list of recommendations separated by commas and spaces. For example:

```
quarrelled -> quarreled, quarreled
```

As you will notice in the example above, it is okay if your program sometimes prints repeat corrections.

If you did not find any candidates in the word_list, meaning you have no good guesses for what the user might have meant, in place of suggestions, print the string `(No suggestions)`, like so:

```
teaparty -> (No suggestions)
```

Be aware that testing three heuristics on dozens of misspellings can take a long time! This is the first time you will have written code that may take minutes rather than seconds to run. This is why we have you start with `declaration.txt` , which should run in a matter of seconds, even on relatively slow computer hardware.

Sample outputs are as follows. The first output is printed in full. The other two are just excerpts:

```
$ python spell.py
Input file: declaration_clean.txt
Word list file: word_list_small.txt
states -> tates, state, stapes, stases, status, stated, stater
events -> evens, event
bands -> band, banns, banda, bande, bandi, bando, bandy
powers -> power
sttaion -> station
laws -> las, law, taws, yaws, lars, lass, lawk, lawn
naturev -> nature
opinions -> opinion
requires -> require, requirer
theg -> the, them, they
causes -> cause, causse, caules, causus, causer, causey
truths -> truth, truthy
selfevident -> (No suggestions)
created -> create, cerated, crested
endowed -> endower
rights -> right, nights, tights, righto, righty
happinessthat -> (No suggestions)
rights -> right, nights, tights, righto, righty
governments -> government
instituted -> institute, instituter
deriving -> driving
powers -> power
governed -> (No suggestions)
ends -> ens, end, enos
rigvht -> right
enw -> new, end, ens
principles -> principes, principle
organizing -> (No suggestions)
powers -> power
governments -> government
changed -> change, changer
causes -> cause, causse, caules, causus, causer, causey
shewn -> hewn, sewn, shen, shawn, shown, shean, sheen, shewa
evils -> evil
ot -> t, o, to, it, of, on, or
abolishing -> (No suggestions)
forms -> form, forme, formy
abuses -> abuse, abusee, abuser
usuppations -> (No suggestions)
pursuing -> (No suggestions)
evinces -> evince
guards -> guard, guardo
futrue -> future
securitysuch -> (No suggestions)
has -> as, ha, bas, das, gas, las, mas, ras, vas, was, yas, his, had, hag, hah, hak, hal, ham, han, hao, hap, hat, ha
colonies -> (No suggestions)
constrains -> constrain, constraint
systems -> system
presyent -> present
injuries -> (No suggestions)
usurpations -> usurpation
having -> caving, paving, raving, saving, waving, hading, hazing
states -> tates, state, stapes, stases, status, stated, stater
facts -> acts, fact, facks, facty
submitted -> submitter
```

```
$ python spell.py
Input file: alice_clean.txt
Word list file: word_list.txt
vii -> oii, vai, vei, via, vic, vie, vim, vip, vis
teaparty -> (No suggestions)
```

```
having -> caving, paving, raving, saving, waving, hading, hazing
using -> sing, suing
elbows -> elbow, elbowy
looked -> booked, hooked, nooked, locked, looker
remarked -> remarker, remarket
isnt -> ist
invited -> invite, invised, invitee, inviter
wants -> want, pants, wanty
hte -> te, he, the, het, ate, ute, hie, hoe, hue
remarks -> remark
... (hundreds of additional "misspelled" words)
```

```
$ python spell.py
Input file: brandeis_clean.txt
Word list file: word_list.txt
brandeis -> (No suggestions)
defendants -> defendant
convicted -> (No suggestions)
persons -> person, persona
charged -> charge, chargee, charger
arrested -> arrestee, arrester
indicted -> indictee, indicter
telephones -> telephone, telephoner
means -> mans, mean, jeans, meant
communicated -> communicate, communicatee
... (hundreds of additional "misspelled" words)
```

Part B: Using a Set

As you have no doubt noticed, this spell-checker takes a very long time to run on `alice.txt` and `brandeis.txt`. In the next two tasks, you will apply some enhancements to allow your spell-checker to run more quickly.

When you loaded the list of words in the word_list from the text file where it was stored, you probably wrote something like the following:

```
words = open(input_filename).read().split('\n')
```

The variable `words` stores a list of strings containing each word in the word_list. To check whether a word is in the word_list, you probably used the boolean test:

```
if word in words:
    ...
```

Unfortunately, this approach has a substantial drawback: checking whether an item is in a list using the `in` operator forces Python to check every item in the list one-by-one. `word_list.txt` contains hundreds of thousands of words, meaning that this operation is exceedingly slow.

Thankfully, we can avoid this problem by using a Python dictionary instead. To do so, you should create a set as described in section 10.3 of the textbook. The variable `words` should store a dictionary where each key is a word in the word_list. The values in this dictionary don't matter; you can simply set the values equal to the boolean value `True`.

```
words_list = open(input_filename).read().split('\n')
words = {}
for word in words_list:
    words[word] = True
```

Your boolean checks of the form `word in words` can remain the same, since this will check whether `word` is a key in the dictionary `words`.

This approach should speed up your program dramatically. Here are results from one computer:

File	word_list	Running Time (word_list as list)	Running Time (word_list as set)
\`declaration_clean.txt\`	\`word_list_small.txt\`	0.289 seconds	0.070 seconds

`alice_clean.txt`	`word_list.txt`	2 minutes 3 seconds	0.177 seconds
`brandeis_clean.txt`	`word_list.txt`	8 minutes 36 seconds	0.221 seconds

Why does this approach work so well? Lists and dictionaries organize their information in different ways. Lists are ordered, whereas dictionaries are relational, storing values in relation to their corresponding keys. Under the hood, Python optimizes each of these datastructures with these qualities in mind. Lists are optimized for accessing based on order - it should be as fast as possible to get element *n* of a list. Dictionaries are optimized for accessing based on key - it should be as fast as possible to get the value corresponding to key *k*. This means that the `in` operator for a list is very slow since lists aren't optimized for finding specific elements by their value. In contrast, the `in` operator for a dictionary is very fast, since dictionaries are already optimized for finding elements by their keys.

In this instance, we are using a dictionary for these fast key lookups without actually using its values at all. This kind of datastructure is known as a *set*. A set keeps track of a collection of elements and makes it fast to check whether any particular element is currently in the set. Here, we are using the dictionary's keys as a set; its values don't matter.

Part C: Adding a Cache

IMPORTANT NOTE: This part of the assignment will be worth a minimal portion of your overall score, so focus all of your attention on the other parts of the assignment until you are confident that everything else is working and do not be too concerned if you are unable to get to this part.

The process of generating every possible correction for a misspelled word generates hundreds of candidates, taking a small amount of processor time that can add up for very long texts. Your program should try to avoid this work whenever possible. Whenever your program attempts to correct a misspelling and generates these candidates, it should save the fruits of this labor to a dictionary, where the key is the misspelling and the value is the string containing the candidates separated by commas and spaces. When your program sees this misspelling in the future, it should retrieve the candidates from this dictionary rather than generating them again from scratch. The idea that we should save previous computation so we don't have to do it a second time is known as *dynamic programming*, and the dictionary you create in the process of doing so is called a *cache*. When you report a misspelling for the second or subsequent time using the cache (what is known as a "cache hit"), print an asterisk after the word in your output to indicate that the cache was used.

For example, the Declaration of Independence example includes the word "rights" twice (see the epilogue for why this is flagged as a misspelling). The second time your program comes across it, it should print:

```
rights* -> right, nights, tights, righto, righty
```

Sample outputs are as follows for `declaration_clean.txt`:

```
$ python spell.py
Input file: declaration_clean.txt
Word list file: word_list_small.txt
states -> tates, state, stapes, stases, status, stated, stater
events -> evens, event
bands -> band, banns, banda, bande, bandi, bando, bandy
powers -> power
sttaion -> station
laws -> las, law, taws, yaws, lars, lass, lawk, lawn
naturev -> nature
opinions -> opinion
requires -> require, requirer
theg -> teg, the, gheg, thig, thug, thea, theb, thee, them, then, theo, thew, they
causes -> cause, causse, caules, causus, causer, causey
truths -> truth, truthy
selfevident -> (No suggestions)
created -> create, cerated, crested
endowed -> endower
rights -> right, nights, tights, righto, righty
happinessthat -> (No suggestions)
rights* -> right, nights, tights, righto, righty
governments -> government
instituted -> institute, instituter
deriving -> driving
powers* -> power
governed -> (No suggestions)
```

```
ends -> ens, end, enos
rigvht -> right
enw -> en, new, end, ens
principles -> principes, principle
organizing -> (No suggestions)
powers* -> power
governments* -> government
changed -> change, changer
causes* -> cause, causse, caules, causus, causer, causey
shewn -> hewn, sewn, shen, shawn, shown, shean, sheen, shewa
evils -> evil
ot -> t, o, to, at, it, st, ut, od, oe, of, og, oh, ok, om, on, or, os, ow, ox
abolishing -> (No suggestions)
forms -> form, forme, formy
abuses -> abuse, abusee, abuser
usuppations -> (No suggestions)
pursuing -> (No suggestions)
evinces -> evince
guards -> guard, guardo
futrue -> future
securitysuch -> (No suggestions)
has -> as, ha, bas, das, gas, las, mas, ras, vas, was, yas, his, had, hag, hah, hak, hal, ham, han, hao, hap, hat, ha
colonies -> (No suggestions)
constrains -> constrain, constraint
systems -> system
presyent -> present
injuries -> (No suggestions)
usurpations -> usurpation
having -> caving, paving, raving, saving, waving, hading, hazing
states* -> tates, state, stapes, stases, status, stated, stater
facts -> acts, fact, facks, facty
submitted -> submitter
```

Adding a cache speeds up `brandeis_clean.txt` from about 0.221 seconds to 0.185 seconds.

Epilogue

You might be underwhelmed by the accuracy of your spell checker. Remember, we're not asking you to build anything that could compete with what Microsoft and Google have developed. But why is your program flagging so many obviously correctly spelled words as potential misspellings?

The biggest culprit is the word_list file we've given you. To save space, Apple has distributed a word_list file that does not include every possible grammatical variation of each word. If it had, the word_list file would have ballooned in size.

To save space, programs that use Apple's word_list are expected to know a few rules of grammar -- how to transform English words into other variations. Once again, these programs are full of heuristics, rules-of-thumb (albeit pretty precise ones) that transform word roots into other valid English words. For example, notice that Apple's word_list evidently does not realize that past-tense verbs can be formed by adding `ed` to many present-tense verbs, hence the identification of `convicted`, `arrested`, and `indicted` as misspelled.

Your program might also leave something to be desired in the way it creates lists of alternative spellings. Microsoft Word and Google Docs tend to offer shorter suggestion lists than your program, and they tend to order suggestions based on what they most think you meant to spell. In fact, production spell checkers take into account not only the misspelled word but the surrounding linguistic context, trying to use pattern matching to figure out what word you probably meant.

Even though the performance of your spell checker might seem limited, it does not operate that differently from the way production spell checkers do at their core. Think about how far you've come in about a month in this course, from knowing nothing about computer programming to being able to create a functional, useful, and user-friendly piece of software!

This problem set was developed by Paul Ohm in 2018 - © 2018