THE UNIVERSITY OF ARIZONA.
DEPARTMENT OF COMPUTER SCIENCE

CSc 120: Writer Bot

Restrictions

1. For the long problem, your code should follow the style guidelines for the class. You must follow the updated guidelines for commenting classes.
2. You may not use concepts and/or short-hand syntax not yet covered in class. The restrictions include the following:
    - dictionary or list comprehensions, e.g., [n * 2 for i in range(10)]
    - with open (explicitly open and close the file instead)
    - the ternary operator (use an if instead)
    - recursion (not allowed for writer-bot)
    - nested functions (using def *within* a function to define another function)
    - exceptions (try/except)
    - type annotations
    - lambda expressions
    - generators and user defined iterators
    - default arguments
    - importing libraries (unless a library is explicitly mentioned in the specification))

Background

Suppose that you are given the task of generating random English text that is—at least somewhat—coherent and readable. You certainly couldn't simply pick random words from a dictionary, since the choice of the parts of speech (nouns, verbs, adverbs, etc.) would be random and the result would be a nonsensical string of words. Likewise, if you randomly selected words from a book, you will have narrowed the choice of words to the vocabulary from the book, but if there is no information on how the words should be sequenced so that the generated text mimics the structure of the English language, the result would again be nonsense.

However, if you could base the generated text on the characteristics of an original, known text, and compute statistics on the text such as the frequency of combinations of words, then the resulting text would replicate those characteristics. In other words, a statistical model of how words are used within a *known* text can be used to generate random text that will have similar statistics as the orginal text. The resulting text would be (potentially) readable and coherent.

To accomplish this, we need a method to analyze text that produces statitics on how combinations of words are used within that text. Fortunately, a method called *Markov Chain Analysis* does precisely that. It determines the probability that certain words will follow combinations of other words in a text. Using Markov chain analysis statistics to generate new text ensures that the new text will have similar statistical properties of the original. (Think of this assignment as a begginer's ChatGPT.)

The theory of Markov chain analysis can be found here.

Markov Chain Algorithm

The Markov chain analysis groups sequences of words into *prefixes* (of a specified size) and determines the set of words that will follow each prefix. A word that follows a prefix is a *suffix*.

For example, consider the lyrics of the Monty Python song *Eric Half-a-Bee*:

> Half a bee philosophically
> Must, ipso facto, half not be.
> But half the bee has got to be
> Vis a vis, its entity. Do you see?
>
> But can a bee be said to be
> Or not to be an entire bee
> When half the bee is not a bee
> Due to some ancient injury?

We can build a table of all of the possible two word prefixes and the suffixes that follow. Since the resulting table is quite large, we will only show the table for a few of the prefixes that help to illustrate the discussion:

Prefix	Suffixes
Half a	bee
a bee	philosophically be Due
bee philosophically	Must
philosophically Must,	ipso
Must, ipso	facto

We can see that the phrase "Half a" is always followed by "bee", but "a bee" can be followed by "philosophically", "be", or "Due".

To generate the text, the Markov chain algorithm will construct phrases by randomly choosing one of the suffixes that follows a given prefix, according to the table that is generated from the text.

For prefixes of length two, the algorithm can be described in pseudocode as follows:

```
create an empty list tlist for the generated text
set w1 and w2 to the first two words in the text
add w1 and w2 to tlist
set the prefix to w1 w2
while the prefix is in the table
   randomly choose w3, one of the successors of prefix w1 w2 in the text
   append w3 to tlist
   set the prefix to w2 w3
```

To illustrate, the algorithm will start by adding "Half a" to `tlist`. The only option for a suffix is "bee", which is then appended to `tlist`. The current prefix changes to "a bee" and the loop repeats. This time, there are three options for suffixes: "philosophically", "be", or "Due". If we suppose that "Due" is chosen, then "Due" is appended to `tlist` and the prefix changes to "bee Due". The generated text in `tlist` at this point is:

```
   [ 'Half', 'a', 'bee', 'Due' ]
```

The text generation continues until the last suffix is reached, or until a sufficient amount of text has been generated. (This is explained further in the **Expected Behavior** section.)

Your program will read a file from input, use the Markov chain analysis to create a table of prefixes and suffixes, and use the pseudocode above to generate new text based on the table of prefix and frequencies.

Definitions

Word

In this problem, we want to keep the punctuation. We want "hurried" to be distinct from "hurried!" so that the generated text will retain some of the grammatical information of the original. A word is therefore defined as a sequence of characters surrounded by white space.

NONWORD

Notice that the generated text must start with "Half a", since those are the first two words of the text. But to build the table, every word must have a prefix. The prefixes for "Half" and "a" at the beginning are boundary cases that would need to be considered in the algorithm. However we can avoid complicating the algorithm to handle these boundaries cases by introducing an artificial word that will never be encountered in the text. We'll define this as NONWORD and we will prime the first two prefixes to be "NONWORD NONWORD" and "NONWORD Half " . Our partial table from the example above would become the following:

Prefix	Suffixes
NONWORD NONWORD	Half
NONWORD Half	a
Half a	bee
a bee	philosophically be Due
bee philosophically	Must
philosophically Must,	ipso
Must, ipso	facto

Multiplicity

The table shown above is only a portion of the full table that would be generated for the example text. In this portion of the table, each prefix/suffix pair occurs only once in the text. For these pairs, we say that the suffix has a multiplicity of 1. However, in the complete table (not shown), a prefix/suffix pair may occur many times. For example, 'bee' is a suffix of 'half the' twice in the text. Since the prefix/suffix pair occurs twice, we say that the suffix has a multiplicity of 2. Likewise, if a pair occurs 4 times, we say that the suffix has a multiplcity of 4.

During text generation, if a suffix has a higher multiplicity, it has a greater chance of being chosen. This means the statistical properties of the original text are maintained.

Expected Behavior

Write a program, in a file **writer_bot.py**, that generates random text from a given source text. Your program should behave as follows:

1. Use **`input()`** (without arguments) to read the name of the the source file *sfile*. Do *not* prompt the user for input. Do *not* hard-code the file name into your program.
2. Use **`input()`** (without arguments) to read in the prefix size *n*. Do *not* prompt the user for input.
3. Use **`input()`** (without arguments) to read in the number of words to be generated for the random text. Do *not* prompt the user for input.
4. Read *sfile* and build the Markov chain table of prefixes to suffixes according to the description above.
5. Construct the randomly generated text according to the Markov chain algorithm. Construct a list to hold the words of the generated text.
6. Print out the generated text list accoring to the **Output format** below.

Input Format

Each line of the input file is a sequence of characters separated by whitespace. The file may consists of any number of lines with any number of words on each line.

Output Format

Print out the list of generated text *ten words per line*. Any extra words will be printed on the last line. For example, if the generated text has only nine words, the output will consist of one line of nine words. If the text has 109 words, the output will consist of eleven lines of output, the first ten lines having ten words and the last line having nine.

Programming Requirements

1. The example discussed above shows a table for prefixes of size two. *Your program must work for a prefix of arbitrary size n.*

2. Use a dictionary to build the table mapping prefixes to suffixes. Since the prefixes will be the keys in a dictionary, you must use an immutable type for the prefixes. *You are required to use tuples for the keys.*

3. As shown in the example, a prefix may have one or more suffixes. You must use a list to represent the possible suffixes. When a new suffix is encountered for an existing prefix, you must append the new suffix to the end of the list. This is important for matching the auto-graded output: the order in which suffixes are stored in the list will affect the choices made during text generation and will impact the output. The following is a snippet of the dictionary corresponding to the Eric the Bee example:

```
{
    ('Half', 'a')  : [ 'bee' ],
    ('a', 'bee')   : [ 'philosophically', 'be', 'Due' ],
    ...
    ('half', 'the'): [ 'bee', 'bee' ],
    ('the', 'bee') : [ 'has', 'is' ]
    ('bee', 'has') : [ 'got' ]
    ...
}
```

   Notice that "bee" occurs twice as a suffix to "half the". Make sure that you keep such duplicates in the suffix list. This ensures that the statisical properties of the text will be correct.

4. You must use the algorithm described in the **Markov Chain Algorithm** section above. If you simply iterate through the keys of the dictionary, your generated text will be incorrect.

5. During text generation, when a prefix has more than one suffix, the suffix will be randomly chosen from the list. You will use the Python random number generator as in Assignment 1 to do this. As in that assigment, in order for your output to match the tester and grading scripts, you must seed the random number generator. To do this, define the following constant at the top of your program:

```
SEED = 8
```

   Note: You must use **`randint()`** and not **`choice()`**. *Also, only call **`randint()`** when there is more than one suffix in the list.*

6. You must define the constant `NONWORD`, which must be a word that cannot exist in the original text. Since a word cannot contain a space, define `NONWORD` as a string with a single space as follows:

```
NONWORD = " "
```

7. As you can imagine, when generating the output for larger text, it is not useful to print out the random text one word at a time. During the text generation phase, create a list to hold the words of the generated text. When the text generation is complete, print the output as specified in the **Output format** section.

Errors

No error checking is required for this assignment.

Examples

Some examples of generating random text from different source texts are shown here.

Reference

The Markov chain algorithm is used to solve a variety of problems. Using it for random text generation has been described in many places, most notably in *The Practice of Programming*, by Kernighan and Pike, which can be found here.
