Generating sets of words

Previously we saw how to generate strings of letters based on some training data that was strings of letters 'A', 'B' and 'C'. This established the approach, but ultimately produced meaningless strings like "BABAACCAAC".

Here we are going to use a set of training data that is sentences, and rather than working letter-by-letter, we are going to work word-by-word. The aim is to produce sentences that look (superficially) like they belong in the source material.

Source material

Since we are producing nonsense, we are going to use a nonsense poem as our input. The Walrus and the Carpenter is a poem by Lewis Carroll that appears in *Through the Looking-Glass, and What Alice Found There* (1871).

First download the source file walrus.txt and save it in the same folder as your Python file. This is the Lewis Carroll poem converted to lowercase characters with the punctuation taken out.

We will read this into Python and `.strip()` the whitespace off the end of each line, storing all the lines in a variable `lines`.

```
lines = []
f = open("walrus.txt","r")
for line in f:
    line = line.strip()
```

```
    lines.append(line)
```

To see the text of the original poem, we can loop through `lines` printing each item.

```
for line in lines:
    print(line)
```

String processing

Each string is a line of the poem and we'll treat this as an item of training data. So instead of items of training data like `"ACB"`, we have items of training data that are strings like `"the eldest oyster winked his eye"`.
When we worked letter-by-letter, we worked out way through the string one character at a time, first `0` then `1` and so on. Here we don't know how long each word is, so we cannot simply slice from the string.

Instead we must split the string up wherever there is a space. Here we use `split` to break each sentence into a list of words.

```
for line in lines:
    words = line.split(" ")
```

Rather than produce sentences of a fixed number of words, we are going to use our program to predict when a line should end. To do this, we have to put an end of line character into the training data. Then when our model decides that the end of line should come next, we will end the line. We can use any string that isn't in the training data, so here we use `"EOL"` for 'end of line'. We add this to the end of the list of words representing each line.

```
for line in lines:
    words = line.split(" ")
    words.append("EOL")
```

Now that we have a list of words on this line, we need to think about what we'd

like to use as training data. We are going to look at two words and use this to predict the next word, so we need to store strings of three words.

Starting from the `i`th item in `words`, we can form a string containing the first two words of a trio using `" ".join(words[i:i+2])`.

We set up a dictionary `starts` to hold all our data. We work through `words` from item `0` until we are two from the end - we find the number of items in `words` using `len(words)` and subtract 2. Then make a new dictionary for every pair of words which counts what they are followed by. We have to be careful to make sure the current value of `start` is not already an item in our dictionary.

Now we have `starts[start]` we use it to count what the third word is in this trio. Again, we use an `if` statement to check if it is already in our dictionary.

```
starts = {}

for line in lines:
    words = line.split(" ")
    words.append("EOL")

    for i in range(0,len(words)-2):
        start = " ".join(words[i:i+2])

        if start not in starts.keys():
            starts[start] = {}

        if words[i+2] not in starts[start].keys():
            starts[start][words[i+2]] = 1
        else:
            starts[start][words[i+2]] += 1
```

Now we can give an opening pair of words and see what the possibilities are for this to be followed by. For example:

```
print(starts["this was"])
```

This tells us the options for following `"this was"` and how many times each one occurred in the source data.

Making a prediction

We're going to make weighted and unweighted choices from lists, so start by importing `choice` (unweighted) and `choices` (weighted) from the `random` package.

```
from random import choice, choices
```

Now for every pair of starting words we can make a list of possible next words and their weights. For example, here we look at the words following "a pleasant".

```
next_words = list(starts["a pleasant"].keys())
weights = list(starts["a pleasant"].values())
next_word = choices(next_words,weights)[0]
print(f"a pleasant {next_word}")
```

What we want to do is continue our sentence until we get an `"EOL"` indicating the end of the line. We can do this with a `while` loop.

Here we store the output in a list called `output`, formed by splitting our input words apart. Then we loop while the last word in `output` is not `"EOL"`. We join the last two words of `output` to form our `start`, then make a list of possible `next_words` and their `weights`, using `choices` to make a selection. Finally we append the word selected to `output` and loop again.

```
output = "and all".split(" ")
while output[-1] != "EOL":
    start = " ".join(output[-2:])
```

```
    next_words = list(starts[start].keys())
    weights = list(starts[start].values())
    next_word = choices(next_words,weights)[0]

    output.append(next_word)
```

You can look at what was produced using

```
print(output)
```

but since this is a list the output isn't very pretty. Ideally we'd like to remove the last item (`"EOL"`) and use `join` to form a sentence.

```
print(" ".join(output[:-1]))
```

Generating a poem

To generate a line of text, we must start with a pair of words that was present in our training data. We can get a list of all the possible starting pairs.

```
possible_starts = list(starts.keys())
```

Now we can use a `choice` from `possible_starts` to start our line, instead of using "and all" every time.

```
possible_starts = list(starts.keys())
output = choice(possible_starts).split(" ")
```

We can use a `for` loop to generate a poem of, say, 30 lines.

```
for i in range(0,30):

    output = choice(possible_starts).split(" ")
```

```
    while output[-1] != "EOL":
        start = " ".join(output[-2:])

        next_words = list(starts[start].keys())
        weights = list(starts[start].values())
        next_word = choices(next_words,weights)[0]

        output.append(next_word)

    print(" ".join(output[:-1]))
```

A note on the quality of the output

We have written a program that will produce gibberish that sounds a bit like lines from The Walrus and the Carpenter.

Because the training data set is so small, a lot of the time the program is just reproducing lines or partial lines from the original poem. If we gave it more text, it would have more options for each word pair. Sometimes it is producing original gibberish, though. For example, mine generated this line

> middle of the night is fine the walrus and the carpenter

At the start, this is following the original line "the middle of the night", but then when it gets to "the night" it has a choice between ending there or switching to the line "the night is fine the walrus said", and it switches. Finally, when it gets to "the walrus", it has the choice of "the walrus and the carpenter" (appears twice in the poem), "the walrus did beseech" (appears once), or "the walrus said" (6 occurrences). It chooses the first of these and then ends the line.

Note that nonsense poems are supposed to make pleasing use of rhythm and rhyme, not just produce gibberish!