Computer Science CS134C (Fall 2018)
Duane A. Bailey
Laboratory 6
Building an Oracle (due Thursday/Friday)

Objective. To build a simple class that generates text in an intelligent (?) manner.

This week we'll complete a simple little class, an `Oracle`, that can be trained to generate "readable" random text. The class makes use of a technique that *fingerprints* a source text, or *corpus*, by keeping track of a distribution of combinations of n letters, called *n-grams*.

The Concept. The central component to this lab is the `Oracle` object. It has the ability to scan and internalize a source text and then, later, it can generate random text that is remarkably similar to the source.

Internally, the `Oracle` keeps track of all the n-grams that appear in a text. For example, the 11-character text

```
'hello world'
```

contains the following nine 3-grams:

```
'hel' 'ell' 'llo' 'lo ' 'o w' ' wo' 'wor' 'orl' 'rld'
```

If a text contains m characters, it is made up of $m-n+1$ n-grams. For a very large text, of course, some n-grams appear quite frequently, while others do not. The power of the `Oracle` class is the development of a *distribution* of n-grams that, in a sense, acts like a "fingerprint" for the author of the text. If we are given $n-1$ letters, we can use the distribution to make an informed guess as to how the author would add a final character to complete an n-gram. When we iterate this process, it is possible to generate text that takes on some of the characteristics of the prose used to train the `Oracle`.

In this week's class definition, we'll keep track of the distribution of n-grams using a dictionary. Each *key* of the dictionary is an $n-1$-character string that is the prefix for one or more n-grams encountered in the text. The *value* associated with the key is a string of all the characters that, when appended to the key, form an n-gram from the corpus. Each character in the value represents one of the n-gram occurances and the distribution of characters that appear in the value reflects the distribution of n-grams that begin with the $n-1$-letter key.

Required Tasks. Here are the steps to completing this week's lab.

1. Download the starter kit for this package in the usual manner:

   ```
   git clone ssh://22xyz9@davey.cs.williams.edu/~cs134/22xyz9/lab6.git lab6
   ```

   This kit is fairly minimal. It contains a single python file, `oracle.py` and several source texts.

2. Run some simple experiments with the function `random.choice` (the `choice` method from the `random` package). This function takes a sequence (a `list`, `tuple`, or `string`) and chooses one of the elements randomly and *uniformly*. Of course, by repeating elements in the sequence, we can simulate any distribution we wish. We'll have to think about how we might use `choice` with non-sequence objects.

3. Examine the `Oracle` class, found in the file `oracle.py`. When completed, the `Oracle` object can be used in the following manner:

```
o = Oracle(n=5)
text = ' '.join([line.strip() for line in open('tomsawyer.txt')])
o.scan(text)
for line in o.lines():
    print(line)
```

The initializer for the `Oracle` takes an n-gram size, `n`. When the `Oracle` uses `scan` to analyze a corpus of text, it reads the text from beginning to end keeping track of the frequency of each combination of `n` characters. This distribution fingerprints the corpus.

The `lines` method generates new lines of text from the distribution seen during the scanning process. The new lines, though random, have the same fingerprint as the original training text.

The remaining steps take us through the process of building a complete `Oracle` class.

4. Observe the `__slots__` attribute. This is a list of the attributes that will hold the state of the `Oracle`. The intent is that the slots support the `Oracle` in the following manner:

   - The `_n` attribute. This attribute is an integer that describes the size of the n-gram window used in scanning the text. It should be 2 or greater, and is determined by the `n` parameter to the initializer.
   - The `_corpus` attribute. This is a copy of all the text that was scanned to develop the textual fingerprint. It is a string of zero or more characters, and is augmented with the `scan` method.
   - The `_dist` attribute. This is a dictionary that keeps track of the distribution of n-grams encountered during the scanning of the corpus. If the n-gram window width is `n`, the keys are the first `n-1` characters of any window seen, and the value is a string that contains *all the possible single character completions encountered*. Because these completions are not uniformly distributed across the character set, there may be many copies of the most common completion letters, and very few copies of the least common completions.

   Read through the `__init__` method. This method accepts an n-gram window size and is responsible for setting all of the slots to a consistent state that represents an empty distribution. There is, after `__init__`, no real fingerprint.

5. We will write, together, the method, `scan(self,s)`, that takes a string, `s`, and records the occurrences of `n` character combinations, where `n` is the n-gram size of the `Oracle`, `self`. Think about how the scanning of `s` should update the other slots of the object. It's important that every method leave the `Oracle` in a consistent state.

6. Write a *private helper method*, `_randomChar(self)` that returns a character at random from the corpus. While every character of the corpus has equal opportunity for being picked, notice that the distribution of characters returned reflects the distribution of the characters that appear in the original text. Since the method begins with an underscore, it will not appear in the documentation for the object; the intent is that this method is only available for use within the object's other methods.

7. Write a private helper method, `_randomKey(self)`, which returns a key randomly selected from the `_dist` dictionary.

8. Write a private helper method, `_next(self,key)`, that, given the first `n-1` characters of an n-gram (`key`), returns a random character that, according to the distribution, typically follows it. If the `key` has never been seen before, it simply returns a random character from the corpus.

9. Write the special method, `__iter(self)__`. It's used in `for` loops. It is simply a generator that yields random characters that arise by trying to extend a randomly selected context—an initial key—to a full n-gram. After each character is generated, the context slides to include the character.

   You should be able to test your `Oracle` with a loop similar to the following:

   ```
   for c in o:
       print(c)
   ```

   and the characters generated should resemble the initially scanned text.

10. Write a method, `lines(self,min=70, max=80)`. This is also a generator, but it yields whole lines—strings of between `min` and `max` characters. (The defaults for `min` and `max` should be 70 and 80.) After `min` characters have been generated, a line is terminated either when the next space is generated or when the maximum length is met. If the generator is carefully constructed, it can produce lines that appear to be consistent from the end of one line to the beginning of the next.

    If your `Oracle` supports the `lines` method, you should now be able to use the following code to generate pages of text:

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

    To limit the number of lines generated to, say, 25 lines, use:

    ```
    for _,line in zip(range(25),o.lines()):
        print(line)
    ```

    This code is the basis for the `oracle.py` behavior as a script. Make sure you understand why this works!

11. Make sure that you provide appropriate documentation strings for the module, the class, and its methods.

12. At this point you can turn in your `oracle.py` file and receive a reasonable grade.

Pushing onward. If you want to go a bit further in your investigation, you might try to tune the default window size so that it generates reasonable text from typical sources. We've included Twain's *Tom Sawyer* and Austin's *Pride and Prejudice* as reasonable examples. Note that if the window size is too small, there's not enough context to re-generate text similar to the original. If the window size is too large, runs of the original text are reconstructed, but the generator obviously "loses its way" from time to time. You're searching for a happy medium that works for several texts.

For the fullest credit, you might consider writing a helper function, `_entropy(string)` (a private function that resides outside the class), that computes the *Shannon entropy*[1] associated with selecting a random character from `string`. The Shannon entropy measures the "unexpectedness" of a character selected from the string.

You can compute the Shannon entropy, H, using the following formula:

$$H = -\sum_{i=0}^{N-1} p_i \log_2 p_i$$

Here, N is the number of distinct characters that you *could* select in this situation. The value p_i is the probability (a value between 0 and 1) that you will pick character i.

Notice that if you are in a situation where exactly one character has a chance of being selected, $N = 1$, $p_0 = 1$ and $\log_2 p_0 == 0$. The total entropy, H, is 0. There is no surprise; no information is gained by learning what the next character is; the result was obvious. Notice that this is independent of the length of the string.

On the other hand, if there are two characters that might be selected with equal likelihood, we have

$$H = -\sum_{i=0}^{N-1} p_i \log_2 p_i = -\sum_{i=0}^{1} 0.5 \cdot -1 = -(0.5 \cdot -1 + 0.5 \cdot -1) = 1$$

In essence, the Shannon entropy tells us how many yes/no questions we would have to have answered before we could identify the next randomly selected character.

You might think about how `_entropy(string)` might be used to support an `_entropy(self,key)` method in the `Oracle` class. This method would return the entropy of the `Oracle` in the situation where it's considering augmenting the `n-1` character string, `key`, to form an n-gram.

⋆

[1] Shannon entropy was introduced by Claude Shannon in 1948. A mathematician working for AT&T, he was interested in modeling how much information could be transmitted across a communication channel.