**Computer Science CS134 (Fall 2020)**
*Daniel Aalberts, Duane Bailey, & Molly Feldman*
Laboratory 5
*Building an Oracle (due Thursday at 5pm)*

**Objective.** To finish a simple module that generates text in an intelligent (?) manner.

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

**The Concept.** The central component to this lab is the module `oracle`. 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` module 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` module 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 fingerprinted by the `oracle`.

In this week's module 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 individually appended to the key, form an $n$-gram from the original text. Each character in the value represents one of the $n$-gram occurrences and the distribution of characters that appear in the value reflects the distribution of $n$-grams that begin with the $(n-1)$-letter key. For example, the distribution dictionary for the `'hello world'` 3-grams would be:

```
{'he': 'l',
 'el': 'l',
 'll': 'o',
 'lo': ' ',
 'o ': 'w',
 ' w': 'o',
 'wo': 'r',
 'or': 'l',
 'rl': 'd'}
```

**Code Review.** Let's download and take a look at the code provided.

Download the repository for this package in the usual manner (use your name instead of 22xyz3):

```
git clone https://evolene.cs.williams.edu/cs134-labs/22xyz3/lab05.git ~/cs134/lab05
```

This repository contains a single Python module `oracle.py` and the pre-processed text from Jane Austen's novel, *Pride and Prejudice*. Acquaint yourself with the text and observe Austen's writing style.

As usual, we begin by carefully reading through the existing code. (We hope to take advantage of work that has already been done!)

◇ Run some simple experiments with the function `random.choice`. This function takes a sequence (a `list`, `tuple`, or `string`) and chooses one of the elements randomly and *uniformly*:

```
>>> from random import choice
>>> choice('hello') # 'h' (20%), 'e' (20%), 'l' (40%), 'o' (20%)
'h'
>>> choice(['hello', 'world']) # 'hello' (50%), 'world' (50%)
'world'
```

As we see, by repeating values in the sequence, we can simulate any distribution we wish. This module makes use of `random.choice` in a number of places, so you should be familiar with how it works.

◇ Familiarize yourself with the `islice(seq, n)` iterator from the package `itertools`. This function limits the number of values delivered by iterating across `seq` to `n`. Here are some examples:

```
>>> from itertools import islice
>>> list( islice(range(10), 3) )
[0, 1, 2]
>>> '-'.join( islice('jello world', 5) )
'j-e-l-l-o'
>>> for line in islice( open('PrideAndPrejudice.txt'), 2):
...     print(line.strip())
...
pride and prejudice a novel by jane austen it is a truth universally
acknowledged that a single man in possession of a good fortune must be in want
```

This is especially useful when you are working with infinite iterators.

◇ Now, let's examine the `oracle` module, in `oracle.py`. All the code we write will be included as part of this module.

When completed, we may use the functions of `oracle` interactively in the following manner:

```
from oracle import *
from itertools import islice  # see above

# read a source text as a long string:
```

```
with open('PrideAndPrejudice.txt') as source:
    text = ' '.join([line for line in source])

# analyze the distribution of n-grams
fingerprint(text, n=3)

# generate 20 lines of random text with width <= 70
for line in islice(lines(width=70),20):
    print(line)
```

The `oracle`'s `fingerprint(text, n)` method scans a string from beginning to end, keeping track of the frequency of each combination of `n` characters. The module uses this distribution to fingerprint the string.

The `lines()` method generates new lines of text from the distribution seen during the scanning process. The new lines, though random, can be expected to have the same fingerprint as the original text.

◇ Near the top of the module, observe the three *global* variables, `_text`, `_n`, and `_dist`. These variables hold the state of the `oracle`'s text generator:

- ⋆ The `int _n`. This 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 set by the `n` parameter to `fingerprint`.
- ⋆ The `_text` string. This is a copy of the text used to develop the textual fingerprint. It is a string of at least `_n` characters given to the `fingerprint` method.
- ⋆ The `_dist` dictionary. This keeps track of the distribution of $n$-grams encountered during the scan of `_text`. 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, there may be many copies of the most common completion letters, and very few copies of the least common completions.

Read through the `fingerprint(text, n=4)` method. This method takes a string, `text`, and records the occurrences of `n` character combinations. This one method determines the value of each of the global state variables, `_text`, `_n`, and `_dist` and ensures they're left in a *consistent* state. Since the state variables begin with an underscore (`_`, meaning "private") users of this module should avoid manipulating these values directly.

◇ The module contains three *private* functions. Since these begin with an underscore, they will not appear in the documentation for the module. The intent is that these functions are only available for use within the module's other methods.

- ⋆ The `_randomChar()` function returns a character at random from the text. While every character of the text has equal opportunity for being picked, notice that the distribution of characters returned reflects the distribution of the characters within the `_text` string, itself.
- ⋆ The `_randomKey()` function returns an `_n-1` character key, randomly selected from the `_dist` dictionary. This function, by the way, might be simplified. Think about how you might do that. How would you test the functionality of `_randomKey`?

⋆ The _randomCompletion(key) function, given key of the first _n-1 characters of an $n$-gram, 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 text. (How might we get into a situation where a key has never been seen before?)

**Required Tasks.** This week's work harnesses the above tools to generate infinite streams of characters, words, and lines that have the same $n$-gram distribution as the fingerprinted text. Here's what needs to be done:

1. Please write the generator, chars(). Because it's a generator, it can be used as the domain of iteration in a for loop. It begins by selecting a random key. It then, using an infinite while loop it repeatedly (1) extends the key to form an $n$-gram, (2) yields a character, and then (3) drops a character from the $n$-gram to form a new key.

   Before tackling a text the length of *Pride and Prejudice*, you can test chars() in interactive Python by creating a fingerprint from some text and then using a loop similar to the following:

```
>>> from oracle import *
>>> from itertools import islice
>>> fingerprint("Hey diddle diddle, the cat and the fiddle.",n=3)
>>> for c in islice(chars(), 10):
...     print(c)
...
h
e

c
a
t

a
n
d
>>>
```

   The characters generated should resemble the initially scanned text. Here, for example, the key is, at some point, 'he' which is then extended to include a space, then the key becomes 'e ', extended to become 'e c', etc.

2. Please write the generator, words(maxlen=20). Using the chars() generator in a loop like:

```
for c in chars():
    ...
```

   it collects non-space characters into words. A word is yielded whenever (1) a space is encountered (check with str's isspace() method), or (2) it becomes maxlen characters long.

As you create your generator, test what words would be generated by different input texts. What words would be printed from input text `t`?

```
>>> t = 'yaddayadda'
>>> fingerprint(t,n=3)
>>> for word in islice(words(maxlen=10), 5):
...     print(word)
...
```

What if `t = 'yin yang yin'`?

3. Write a final generator, `lines(width=80)`. This generator yields lines of words from `words()` that come close to but do not exceed `width`. We've given you a start: a list, `wordList`, that collects words used to form the line. If the generator is carefully constructed, it will produce lines that read consistently from one line to the next; it never drops any words from the underlying `words()` generator. You should think about how to limit word lengths; source texts that have no spaces (like `'yaddayadda'`, above) must be handled reasonably.

   If your `oracle` module supports the `lines` method, you should now be able to use the following code to generate an infinite amount of text (use Control-c if you're stuck in an infinite loop):

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

   To limit the number of lines generated to, say, 20 lines, use:

   ```
   >>> for line in islice(lines(), 20):
   ...     print(line)
   ```

   Again: make sure you understand why this works!

4. Run your doctests, review your code, check your documentation, and sign the honorcode in `honorcode.txt`.

5. At this point you can turn in your work for grading. Please make sure you add, commit, and push `oracle.py` and `honorcode.txt`.

**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 Austen's *Pride and Prejudice* and Alcott's *Little Women* as 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. If you *do* experiment with this, document your findings as a suggestion about how to set fingerprint's n parameter in the module's documentation at the top of the file.

For the fullest credit, you might consider writing a helper function, `entropy(string)`, that computes the *Shannon entropy*[1] associated with selecting a random character from `string`. The Shannon entropy measures the "unexpectedness" of the character returned by `random.choice(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 in the string. The value $0 \leq p_i \leq 1$ is the probability 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, $H = 0$. There is no surprise; no information is gained by learning what the next character is; the result was obvious.

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. If you randomly pick a letter from *Pride and Prejudice*, how many yes/no questions would you expect to have to ask to accurately guess the letter? The Shannon entropy of the text will tell us that.

Given `entropy(string)`, we could then write `keyEntropy(key)`. This method measures the "unexpectedness" of the result of `_randomCompletion(key)`. Do you have choices? Then the entropy is greater than zero. On the other hand, if the entropy of the key is zero, the text generator is *reproducing* the text, verbatim.

⋆

[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.