Computer Science CS134 (Spring 2020)
Shikha Singh & Iris Howley
Laboratory 10 – Extra Credit
Building an Oracle (due Thursday May 7, 11pm EST)

Lab Overview Video. The following videos provide an overview to this lab assignment:
https://williams.hosted.panopto.com/Panopto/Pages/Viewer.aspx?pid=f675fd4b-2b35-47e7-a8a6-abae0170bb9a.

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

This week we'll complete a 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.

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

1. Download the starter kit for this package in the usual manner, using your username:

   ```
   git clone https://evolene.cs.williams.edu/cs134-s20/lab10/22xyz3.git lab10/
   ```

   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('prideandprejudice.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. The method, `scan(self,s)`, 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 char in myOracle:
       print(char)
   ```

   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, ideally split on a space character. (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 (whichever occurs first). 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! Please come to instructor or TA Student Help hours if you have questions, hours are posted on the course website.

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

Submitting your work. When you're finished, stage, commit, and push your work to the server as you did in previous labs. Remember that you must certify that your work is your own, by typing out the Honor Code statement in the `honorcode.txt` file, committing and pushing it along with your work.

Late days. You are allowed a total of 3 late days over the semester, with at most 2 late days towards any one lab. You must request a late day in advance on the form, here: `http://bit.ly/s20late`.

Grading Guidelines. Both functionality and programming style are important when writing code, just as both the content and the writing style are important when writing an essay. In this program, some of the specific functional requirements we will test for include:

- Your code should work for any text input given, but for testing purposes, you might try the following test oracle:

```
testOracle = Oracle(n=3)
testOracle.scan('this that and the other thing')
```

  - The length of `_corpus` for `testOracle` should be 29, and the length of its `_distribution` is 19.
  - When called repeatedly, `_randomChar` should only produce 12 unique characters for `testOracle` and `_randomKey` should only produce 19 unique keys.
  - For the `_next` method, when given the key `'th'` it should only ever produce letters from the string `'aei'` as possible n-1 grams.
  - When `_next` is called many times with a key that does not exist, such as `'zz'`, it should produce at most 12 unique n-1 grams.

- Recall that we never explicitly call a special method on a class. There are *implicit* means of calling special methods.
- Consider when and how to use required methods. If you're not using one of the required methods, then you're likely duplicating code somewhere!
- Pay special attention to parameters passed to methods. If we're requiring a parameter, then you should be using it!
- All functions with predictable/testable outputs should have a robust set of doctests that appropriately test your functions. Aim for a minimum of 2 doctests per testable method.
- Just like last previous labs, we require that the functions defined in `oracle.py` follow our specifications, e.g., `scan` takes in two parameters— `self` and a string `s`—in that order and updates internal variables, etc. Do not modify the method names, their parameters, nor what is returned. The examples above in **This week's tasks** provide details about these constraints.
- It is very important that your files be named properly so that we can run our tests for grading. The only acceptable names and capitalization are: `oracle.py`.

Stylistically, we expect to see programs that exhibit the following: meaningful names used in declarations, informative comments (both in-line and docstrings), good and consistent formatting, and good choice of Python commands. There is some subjectivity to what makes good style, but the basic goal is to make your ideas as clear and easy to follow as possible.

Grading scale. Programming labs will be graded on the following scale:

A+	An absolutely fantastic submission of the sort that will only come along a few times during the semester.
A	A submission that exceeds our standard expectations for the assignment. The program must reflect additional work beyond the requirements or get the job done in a particularly elegant way.
A-	A submission that satisfies all the requirements for the assignment – a job well done.
B+	Submission meets the requirements for the assignment, possibly with a few small problems.
B	A submission that has problems serious enough to fall short of the requirements for the assignment.
C	A submission that has extremely serious problems, but nonetheless shows some effort & understanding.
D	A submission that shows little effort and does not represent passing work.