Assignment 7 Python - Text Riffing

Due Mar 24, 2016 by 12pm **Points** 90 **Submitting** an external tool
Available until Mar 26, 2016 at 12pm

This assignment was locked Mar 26, 2016 at 12pm.

- No code is provided.
- **Test** (http://codeskulptor.appspot.com/owltest/?urlTests=comp130.assignment_text_riffing_tests.py&urlPylintConfig=comp130.pylint_config.py)

Put all your answers in one CodeSkulptor file. Put text answers in Python comments. Use Python comments to clearly mark each problem number.

- **(10 points — 5 points automatically checked; 5 points using helper functions to decompose problem)** Code style. Decompose complicated functions into simpler ones. Be sure to use functions from previous exercises and assignments, where applicable.
- **(5 points)** Meaningful docstrings for all functions

Text Riffing

Given a Markov chain, we can generate text based upon it with the following general approach.

1. Pick one of the chain's/dictionary's n-word keys.
2. Initialize a list of strings with these words.
3. Loop as long as desired:
   a. Use the previous n-word sequence to look up possible successors in the chain/dictionary. If there are no successors, exit from the loop.
   b. Pick one of these successors, based upon the probabilities.
   c. Add it to the list of strings.
4. Convert that list of strings into a single string, and return it.

For example, assume we use 2-word sequences, and thus we are using the 2nd-order Markov chain for Eight Days a Week. Note that for this assignment, we are including punctuation in our definition of words.

- Initialization: Assume we randomly pick `('I', 'need')` as our starting sequence. We add `'I'` and `'need'` to an empty list — a list that will eventually contain all our generated text.
- Iteration 1: We see that the 2-word sequence `('I', 'need')` can be followed by `'you'` or `'your'`, each with 0.5 probability. We randomly pick `'you'` and add that to our list.
- Iteration 2: We now see that the 2-word sequence `('need', 'you')` must be followed by a comma, so we add that to our list.
- Iteration 3: We now see that the 2-word sequence `('you', ',')` can be followed by `'Hold'` or `'Eight'`, each with 0.5 probability. We randomly pick `'Hold'` and add that to our list.

- Finishing: Let's assume we only want to generate 5 words in this example. We have generated the list `["I", "need", "you", ",", "Eight"]`, and we can use `string.join()` to get `"I need you , Eight"`.

There's a potentially confusing special case. If the last n words in the original text only appear as that sequence once at the end, then that sequence has no possible successors. So, when generating our text, if we are working with that sequence, we can't possibly continue, so we just exit from the loop.

1. **(25 points)**

   Define the function `random_choice_weighted(choices, random_fn)`, which takes a dictionary, where each possible choice is mapped to its probability. You may assume without checking that the probabilities are non-negative and sum to 1. The function returns one of the choices picked randomly with respect to the weighted probabilities.

   It takes a *function* that takes no argument and returns a value from 0 to 1. Typically, you would pass the function `random.random`. I.e., you would call `random_choice_weighted(…, random.random)`, **not** `random_choice_weighted(…, random.random())`. However, this allows us to also pass non-random functions and get predictable results for testing purposes. For example, you could do the following:

   ```
   def always_zero():
   {
       return 0
   }

   print random_choice_weighted(…, always_zero)
   ```

   Hint: Structure your solution so it works like the following example. Assume your choices are `{"a" : 0.25, "b" : 0.5, "c" : 0.25"}`. We use `random_fn()` to randomly pick a number in the range [0,1), say, 0.8. Our number 0.8 is not less than or equal to the first probability 0.25, so we skip the first option and adjust our number to 0.55 = 0.8 - 0.25. Our number 0.55 is not less than or equal to the next probability 0.5, so we skip that second option and adjust our number to 0.05 = 0.55 - 0.5. Our number 0.05 is less than or equal to the next probability 0.25, so we pick that option and return `"c"`.

2. **(25 points)**

   Define the function `generate_text(chain, num_words, starting_fn, random_fn)`. Given a Markov chain, it generates a random sequence of the given number of words using the algorithm described at the beginning of this section. As a special case, the result can be shorter than the specified number of words if the generation routine happens to comes across a word sequence with no successor. It returns a string of the resulting text.

   The Markov chain input should be created by the code from the previous assignment's last problem. When generating test input, you may use your code or our sample solution.

   It takes two functions. `starting_fn` helps us with the first step of the algorithm. It is a function that takes a list of all the chain's keys and returns one of them to be the starting sequence. `random_fn` is simply passed to the helper function `random_choice_weighted()`. Thus, you could test your function with code like the following:

```
chain = wordseq_successor_frequencies("comp130_EightDaysAWeek.txt", 3, True, True)
print generate_text(chain, 5, random.choice, random.random)
```

Sets and Graphs

3. **(25 points)**

   Define the function `neural_net(input_node_set, num_hidden_nodes, output_node_set)`. It takes two sets and one integer. It returns a directed graph that has the structure described below, which is a slightly simplified version of what we will use later in the course as an artificial neural network.

   You need to generate a set of “hidden nodes” of the specified size. Each input node has an edge to each hidden node. Each hidden node has an edge to each output node.

   In addition, you need to verify that neither input set is empty and that the number of hidden nodes is positive. You also need to verify that the two node sets are disjoint (i.e., have an empty intersection). If any of these tests fail, your function should return `None`.

   We also want the generated names for the hidden nodes to be distinct from that of the input and output nodes. We will assume that the input and output nodes are given as strings, while you generate the hidden nodes as integers from 0 to `num_hidden_nodes`-1.

   As an example, `neural_net(set(["a", "b", "c"]), 2, set(["x", "y"]))` should produce the following:

```
{"a": set([0, 1]),
 "b": set([0, 1]),
 "c": set([0, 1]),
 0: set(["x", "y"]),
 1: set(["x", "y"]),
 "x": set([]),
 "y": set([])}
```