CS61A Lab 6: Nonlocal and Dictionaries and Shakespeare

Week 6, 2011

Nonlocal

Predict the result of evaluating the following calls in the interpreter. Then try them out yourself!

```
>>> def make_funny_adder(n):
        def adder(x):
                if x == 'new':
                        nonlocal n
                        n = n + 1
                else:
                        return x + n
        return adder

>>> h = make_funny_adder(3)

>>> h(5)
...

>>> j = make_funny_adder(7)

>>> j(5)
...


>>> h('new')

>>> h(5)
...

>>> j(5)
```

Write a function `make_fib` that returns a function that returns the next Fibonnaci number each time it is called. See the following examples:

```
>>> fib = make_fib()

>>> fib()
1

>>> fib()
1

>>> fib()
2

>>> fib()
3
```

```
>>> fib()
5
```

List Comprehension

In last week's lecture you were introduced to the concept of mutable data types. Python's list construct is a powerful data type that can be modified once created, unlike tuples. Some operations on lists include indexing, slicing, and extended slicing. Try to guess what the following statements would evaluate to if entered in the interpreter (It may help to draw box and pointer diagrams like those shown in lecture):

```
>>> a, b = [1,2,3], [4,5,6]
>>> a.append(b)
>>> print(a)
...

>>> b[:] = 0,0,0
>>> print(b)
>>> print(a)
...

>>> a[2] = [4]
>>> print(a,b)
>>> print(a[1:3])
...

>>> a[::-1]
...
```

Similar to the generator expressions you've seen previously, lists can be created using a syntax called "list comprehension." Using a list comprehension is very similar to using the map or filter functions, but will return a list as opposed to an iterable.

```
>>> a = [x+1 for x in range(10) if x % 2 == 0]
>>> a
[1, 3, 5, 7, 9]
```

Shakespeare and Dictionaries

First, let's talk about dictionaries. Dictionaries are simply an unordered set of key-value pairs. To create a dictionary, use the following syntax:

```
>>> webster = {'Hamilton': (15, 16), 'Eric T.': (13, 14)}
```

The curly braces denote the key-value pairs in your dictionary. Each key-value pair is separated by a colon, and for each pair, the key appears to the left of the colon and the pair appears to the right of the colon. You can retrieve values from your dictionary by using the key:

```
>>> webster['Hamilton']
(15, 16)

>>> webster['Eric T.']
(13, 14)
```

To modify the entry for an existing key in the dictionary, use the following syntax. Adding a new key follows the identical syntax!

```
>>> webster['Hamilton'] = (11, 12)

>>> webster['Hamilton']
```

```
(11, 12)

>>> webster['Eric K.'] = (17, 19)

>>> webster['Eric K.']
(17, 19)
```

Now that you know how dictionaries work, we can move on to approximating the entire works of Shakespeare! We're going to use a bigram language model. Here's the idea: We start with some word - we'll use "The" as an example. Then we look through all of the texts of Shakespeare and for every instance of "The" we record the word that follows "The" and add it to a list. Then we randomly choose a word from this list, say "cat", and repeat the process. This eventually will terminate in a period (".") and we will have generated a Shakespearean sentence!

The object that we'll be looking things up in is called a 'successor table', although it's really just a dictionary. The keys in this dictionary are words, and the values are lists of successors to those words.

(A copy of the framework code is located in ~cs61a/lib/shakespeare.py)

Here's an incomplete definition of the build_successors_table function. The input is a list of words (corresponding to a text), and the output is a successors table. (By default, the first word is a successor to '.') See this example:

```
>>> def build_successors_table(tokens):
        table = {}
        prev = '.'
        for word in tokens:
                if prev in table:
                        "**FILL THIS IN**"



                else:
                        "**FILL THIS IN**"



                prev = word
        return table



>>> text = 'The', 'cat', 'and', 'the', 'dog', 'both', 'ate', 'and', 'ran', 'outside', '.'

>>> table = build_successors_table(text)

>>> table
{'and': ['the', 'ran'], 'both': ['ate'], 'ate': ['and'], 'ran': ['outside'], 'dog': ['both'], 'cat': [
```

Let's generate some sentences! Suppose we're given a starting word. We can look up this word in our table to find its list of successors, and then randomly select a word from this list to be the next word in the sentence. Then we just repeat until we reach some ending punctuation. (Note: to randomly select from a list, first make sure you import the Python random library with `import random` and then use the expression `random.choice(my_list)`) Here's a definition of `construct_sent` to get you started.

```
>>> def construct_sent(word, table):
        import random
        result = ''
        while word not in ['.', '!', '?']:
                "**FILL THIS IN**"




        return result + word
```

Great! Now all that's left is to run our functions with some actual code. The following snippet will return a list containing the words in all of the works of Shakespeare (warning: do not try to print the return result of this function):

```
>>>def shakespeare_tokens(path = 'shakespeare.txt', url = 'http://inst.eecs.berkeley.edu/~cs61a/fa11/sh
         """Return the words of Shakespeare's plays as a list"""
         import os
        from urllib.request import urlopen
         if os.path.exists(path):
                 return open('shakespeare.txt', encoding='ascii').read().split()
         else:
                 shakespeare = urlopen(url)
                 return shakespeare.read().decode(encoding='ascii').split()
```

Next, we probably want an easy way to refer to our list of tokens and our successors table. Let's make the following assignments:

```
>>>tokens = shakespeare_tokens()

>>>table = build_successors_table(tokens)
```

Finally, let's define an easy to call utility function:

```
>>>def sent():
        return construct_sent('The', table)

>>> sent()
" The plebeians have done us must be news-cramm'd "

>>> sent()
" The ravish'd thee , with the mercy of beauty "

>>> sent()
" The bird of Tunis , or two white and plucker down with better ; that's God's sake "
```

Huzzah!