Python Guide for Ngram Assignment

This guide goes over some Python knowledge that will help you complete the ngram assignment.

Command line arguments

Data passed to Python from the command line is always available, but has to be imported.

```
$ python my_file.py 3 4 hello goodbye
>>> from sys import argv
>>> argv
['my_file.py', '3', '4', 'hello', 'goodbye']
```

The arguments are given in a list of strings (even if a string only has numeric characters), and name of the file is always the first element.

List slicing

Recall that the (mutable) ordered sequence type in Python is named "list". They are similar to the data type called "arrays" in C-style languages, though their length is not fixed. When one talks about "arrays" in Python, it's assumed that they're talking about the array data type defined in Numpy.

Lists have a few properties that make them easy to work with: you can use negative indices to count from the end of the list, and you can use `start:stop:step` syntax to pick out specific slices. You don't have to include all three values, and when one is missing, `start` is assumed to be `0`, `stop` is assumed to be the current length of the list, and `step` is assumed to be `1`.

```
>>> stuff = ['a', 'b', 'c', 'd']
>>> stuff[-1]
'd'
>>> stuff[1:]  # from the second element onward
['b', 'c', 'd']
>>> stuff[:-2]  # up to the next-to-last element
['a', 'b']
>>> stuff[1:-1]  # the middle, without the first and last element
['b', 'c']
>>> stuff[::2]  # every other element
['a', 'c']
```

Hashing tuples

Recall that dictionaries are the hash table implementation in Python. You may find yourself tempted to have nested dictionaries at some point in this assigment, but this is unnecessary and will make your code more difficult to read. You can instead use tuples as dictionary keys and look things up by n-tuples rather than n nested dictionaries.

```
>>> new_dict = {('a', 'b', 'c'): 2, ('b', 'c', 'a'): 5}
>>> new_dict['b', 'c', 'a']
5
>>> new_dict['a', 'b', 'c']
2
>>> new_dict['a', 'b', 'c'] += 1
>>> new_dict['a', 'b', 'c']
3
```

Update: I did actually think of a good potential solution that has one level of nested dictionaries (dict -> dict -> value), regardless of the value of `n`, so don't let this section dissuade you from that if you come up with a solution that does that.

Converting a list to a tuple, and other data structure conversions.

If you have a list, you can make it a tuple with `tuple(...)`

```
>>> stuff = [1, 2, 3]
>>> tuple(stuff)
(1, 2, 3)
```

If you were planning to convert a list of anything other than numeric types or strings (which are immutable), remember that the copying is done by reference, so this could result in unexpected behavior with mutable types.

`list(...)` can do the reverse operation.

`list(some_dict)` returns a list of tuples of the key-value pairs of a dictionary, though you can iterate over a dictionary without converting it to a list.

Counting with special dictionary subclasses

No external libraries are allowed for this assigment, though the standard library has a few (including `re`) that will save you from needing to re-invent the wheel (except for the wheel-reinvention that is the learning objective of the assigment). Specifically, they will help with counting things.

`collections.defaultdict`

`defaultdict` will automatically supply a value if a key is looked up that isn't present. In this case, it would be useful to have a `defaultdict` that automatically supplies `0`.

```
>>> from collections import defaultdict
>>> new_dict = defaultdict(int)  # supply whatever `int()` returns, which is `0`, when a key isn't present
>>> new_dict
defaultdict(<class 'int'>, {})
>>> new_dict['a', 'b', 'c'] += 1
>>> new_dict
defaultdict(<class 'int'>, {('a', 'b', 'c'): 1})
```

`collections.Counter`

`Counter` will solve a lot of the same problems as `defaultdict`, but has more features. One constructs a `Counter` by passing it an iterable of some kind (often a list).

```
>>> from collections import Counter
>>> counts = Counter(['a', 'b', 'a', 'c', 'c', 'a'])
>>> counts
Counter({'a': 3, 'c': 2, 'b': 1})
>>> counts['a']
3
>>> counts['z']  # This hasn't been counted!
0
>>> counts + Counter(['x', 'y', 'x', 'z'])  # You can merge Counters with addition
Counter({'a': 3, 'c': 2, 'x': 2, 'b': 1, 'y': 1, 'z': 1})
```

Repeating something n times

If you want to repeat certain code a certain number of times, don't do this:

```
# bad
num_times = 5
while num_times > 0:
    do_stuff()
    num_times -= 1
```

Do this instead

```
# good
for _ in range(5):
    do_stuff()
```

`range` generates integers from `0` to `n - 1`, though you can indicate that you're just using it to repeat something `n` times by naming the loop variable `_`, which indicates that the value isn't ever going to be used. If you plan to use a variable even once, don't name it `_`.

Joining strings

If you have a list (or some other iterable type) of strings, you can concatenate all of them together with the `join` method of strings. Use that method on whatever string you want to go between each string--often, this will be a string of one single space.

```
>>> ' '.join(['I', 'like', 'nlp'])
'I like nlp'
```

You can also create a stand-alone function that permanently refers to that instance method.

```
>>> space_join = ' '.join  # do not call it, just access it
>>> space_join(['I', 'like', 'nlp'])
'I like nlp'
>>> space_join(['ngrams', 'are', 'cool'])
'ngrams are cool'
```

Sorting lists

Lists have a `sort` method that sorts the list in-place and returns `None`. (This is basically the same as a void method in Java, but Python functions and methods must return *something* to uphold "everything's an object".) So don't do `my_list = my_list.sort()` or you will probably lose your list.

The `sort` method has an optional `reverse` keyword argument that can be `True` or `False` (and is `False` if unspecified). It also has a `key` argument to tell the algorithm how to sort a given element if you want something other than the default sorting behavior.

```
>>> stuff = [('a', 5), ('b', 3), ('c', 7)]
>>> stuff.sort(reversed=True, key=lambda x: x[1])
# `stuff` is now sorted by the second element of each tuple in descending order
>>> stuff
[('c', 7), ('a', 5), ('b', 3)]
```

`lambda` is a reserved word in the language. Without going into the specifics of what a lambda is, in this case, `x` represents any element in the list, and that element gets sorted in terms of whatever the expression after `lambda x:` evaluates too.

Getting a random float between 0.0 and 1.0

```
>>> from random import random
>>> random()
0.46378739448382433
```