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
```

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 `_`.