Motivation

Suppose I had Melville's MOBY DICK stored in a text file called `moby.txt`. What if I was interested in finding the most frequent word used in the text? It's easy enough to hold all of MOBY DICK in memory, so I can read the entire text into a string, split the words using whitespace as my delimiter and produce a list of words, which we call tokens.

```
def file_to_tokens(filename):
    with open(filename) as fin:
        return fin.read().split()
```

Now I'm left with the task of counting the how many times each token occurs in the list. I could use list operations to first find the set of unique tokens, and then count the occurrences of those tokens.

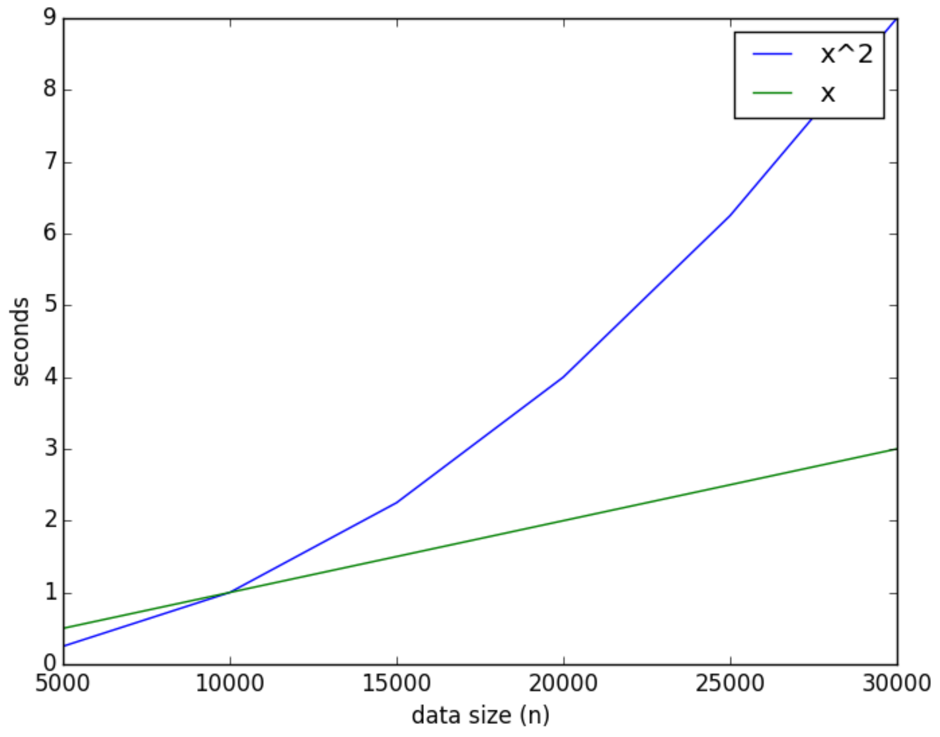
```
def wc_list(tokens):
    uniq = []
    for token in tokens:
        if token not in uniq:
            uniq.append(token)
    return [(t, tokens.count(t)) for t in uniq]
```

Let's think about this. Suppose we have a list of n tokens. For each token in the list, we have to scan the list of unique tokens seen so far to see if it occurs. In the worst case, our set of unique tokens has size that's linear in n, which means that finding the unique tokens takes around n^2 operations. After that, we still have to count the number of occurrences of each unique token in the original token list, which is another n^2 operations.

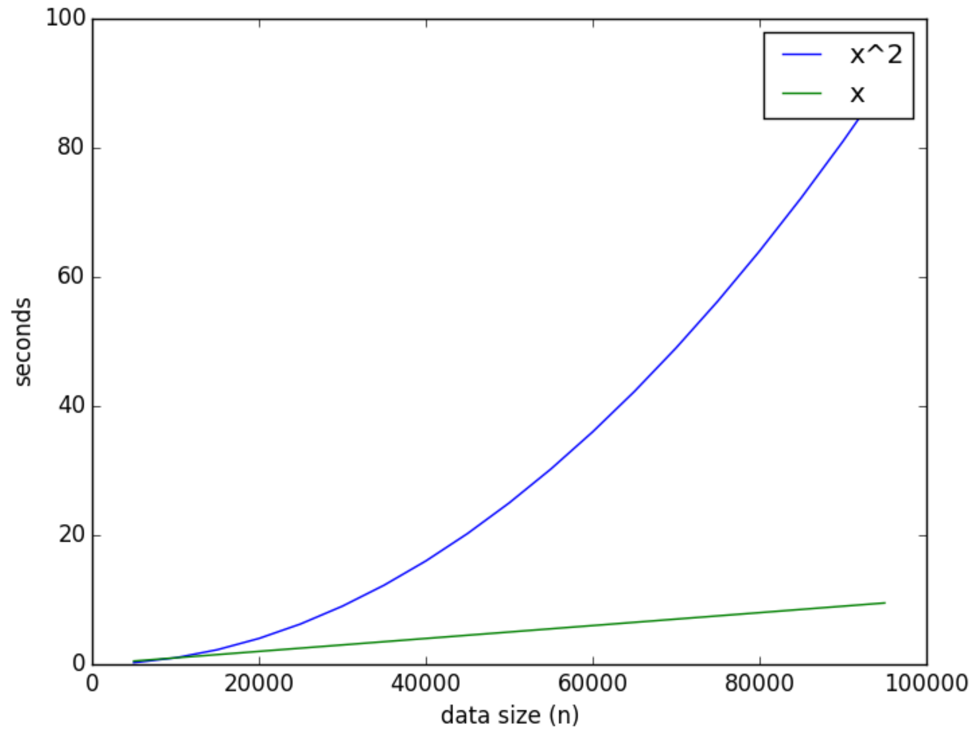
Let's do a back-of-the-envelope calculation. There are around 212,000 tokens in MOBY DICK. Suppose that `uniq` contains a list of the unique tokens. A lower bound on the number of operations is to search `uniq` instead of `tokens` for a count. Searching the first 5000 unique tokens for the first 5000 tokens take half a second.

```
>>>import cProfile
>>> cProfile.run('[uniq[:5000].count(t) for t in uniq[:5000]]')
         5004 function calls in 0.528 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.147    0.147    0.528    0.528 <string>:1(<listcomp>)
        1    0.000    0.000    0.528    0.528 <string>:1(<module>)
        1    0.000    0.000    0.528    0.528 {built-in method exec}
     5000    0.382    0.000    0.382    0.000 {method 'count' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' }
```

We know, however, that this searching grows quadratically, as the number of unique tokens increases. MOBY DICK contains around 35,000 unique tokens so it will only take at least 9 seconds to get the counts.



But for a text with 100,000 unique tokens close to two minutes to find the counts. Thus, lists don't seem viable for even moderately sized texts.



Dictionaries

Ideally, we could scan the list of tokens once and keep a count associated with each token. If we see a token for the first time, we associate a 1 with it. If we see it again, we add 1 to its current count. To only scan the tokens once requires the ability to look up the token count immediately. Dictionaries, more or less, allow us to do just that. Contrast this with our list implementation: determining the count for a single token required scanning the entire list. But dictionaries aren't magic. For them to work requires the ability to take a Python object like a string, integer, or tuple and *hash* it to an integer so that different Python objects rarely collide. We won't worry about these details now, but by choosing a hash function intelligently, one can keep the number of collisions minimal.

Syntax

Dictionaries maintain a set of *key / value pairs*. In Python, the *key* can be anything that is immutable—strings, tuples, numbers, etc. The *value* can be any Python object, including mutable data structures like lists or other hash tables.

Here is the syntax for creating an empty dictionary, adding keys, retrieving the values associated with keys, and checking if a dictionary has a given key.

```
>>> d = {}
>>> d["brent"] = 38
>>> d["courtney"] = 40
>>> d["oscar"] = 5
>>> d["george"] = 1
>>> d
{'oscar': 5, 'courtney': 40, 'brent': 38, 'george': 1}
>>> d["oscar"]
5
>>> "brent" in d
True
>>> "amy" in d
False
```

There are three ways of iterating over the data in a dictionary `d`:

by key `d.keys()` returns a view of the keys of the dictionary that is iterable. In other words `list(d.keys())` will give you a list of the keys in the dictionary.

by value `d.value()` returns a view of the values of the dictionary that is iterable. Similarly, `list(d.values())` returns a list of the values of the dictionary.

by key/value pairs `d.items()` returns a view of the dictionary yielding tuples of the form `(key,value)`. Calling `list(d.items())` returns a list of tuples.

```
>>> list(d.keys())
['oscar', 'courtney', 'brent', 'george']
>>> list(d.values())
[5, 40, 38, 1]
>>> list(d.items())
[('oscar', 5), ('courtney', 40), ('brent', 38), ('george', 1)]
>>>
```

With these operations in hand, we can now construct an efficient version of word count.

```
counts = {}
for token in tokens:
    if token in counts:
        counts[token] += 1
    else:
        counts[token] = 1
return counts.items()
```

Here are some other methods that capture idioms often encountered with a dictionary `d`:

- `setdefault(key, default=None)` If `key` is in `d`, then return `d[key]`, otherwise insert `key` with value `default` and return `default`.
- `get(key, default=None)` If `key` is in `d`, then return `d[key]`, otherwise, return `default`.

Notice that `get` makes our method above more compact.

```
counts = {}
for token in tokens:
counts[token] = counts.get(token,0) + 1
return counts.items()
```

Practice

Suppose we wanted to create an index of the positions of each token in the original text. Write a function called `token_locations` that, when given a list of tokens, returns a dictionary where each key is a token and each value is list of indices where that token appears.

```
>>> l = "brent sucks big rocks through a big straw".split()
>>> print(token_locations(l))
{'big': [2, 6], 'straw': [7], 'brent': [0], 'a': [5], 'through': [4], 'sucks': [1], '
```