Daily Coding Problem #232

Problem

This problem was asked by Google.

Implement a `PrefixMapSum` class with the following methods:

- `insert(key: str, value: int)`: Set a given key's value in the map. If the key already exists, overwrite the value.
- `sum(prefix: str)`: Return the sum of all values of keys that begin with a given prefix.

For example, you should be able to run the following code:

```
mapsum.insert("columnar", 3)
assert mapsum.sum("col") == 3

mapsum.insert("column", 2)
assert mapsum.sum("col") == 5
```

Solution

Depending on how efficient we want our `insert` and `sum` operations to be, there can be several solutions.

If we care about making `insert` as fast as possible, we can use a simple dictionary to store the value of each key inserted. As a result insertion will be `O(1)`. Then, if we want to find the `sum` for a given key, we would need to add up the values for every word that begins with that prefix. If `N` is the number of words inserted so far, and `k` is the length of the prefix, this will be `O(N * k)`.

This could be implemented as follows:

```
class PrefixMapSum:
    def __init__(self):
        self.map = {}

    def insert(self, key: str, value: int):
        self.map[key] = value

    def sum(self, prefix):
        return sum(value for key, value in self.map.items() if
key.startswith(prefix))
```

On the other hand, perhaps we will rarely be inserting new words, and need our `sum` retrieval to be very efficient. In this case, every time we `insert` a new key, we can recompute all its prefixes, so that finding a given sum will be `O(1)`. However, insertion will take $O(k^2)$, since slicing is `O(k)` and we must do this `k` times.

```
from collections import defaultdict

class PrefixMapSum:
    def __init__(self):
        self.map = defaultdict(int)
        self.words = set()
```

```
    def insert(self, key: str, value: int):
        # If the key already exists, increment prefix totals by the difference
of old and new values.
        if key in self.words:
            value -= self.map[key]
        self.words.add(key)

        for i in range(1, len(key) + 1):
            self.map[key[:i]] += value

    def sum(self, prefix):
        return self.map[prefix]
```

A hybrid solution would involve using a trie data structure. When we insert a word into the trie, we associate each letter with a dictionary that stores the letters that come after it in various words, as well as the total at any given time.

For example, suppose that we wanted to perform the following operations:

```
mapsum.insert("bag", 4)
mapsum.insert("bath", 5)
```

For the first `insert` call, since there is nothing already in the trie, we would create the following:

```
{"b": "total": 4,
    {"a": "total": 4,
        {"g": "total": 4}
    }
}
```

When we next insert `bath`, we will step letter by letter through the trie, incrementing the total for the prefixes already in the trie, and adding new totals for prefixes that do not exist. The resulting dictionary would look like this:

```
{"b": "total": 9,
    {"a": "total": 9,
        {"g": "total": 4},
        {"t": "total": 5,
            {"h": "total": 5}
        }
    }
}
```

As a result, `insert` and `sum` will involve stepping through the dictionary a number of times equal to the length of the prefix. Since finding the dictionary values at each level is constant time, this algorithm is `O(k)` for both methods.

```
from collections import defaultdict

class Trie:
    def __init__(self):
        self.letters = defaultdict(dict)
        self.total = 0

class PrefixMapSum:
    def __init__(self):
        self.root = Trie()
        self.map = {}

    def insert(self, key: str, value: int):
        # If the key already exists, increment prefix totals by the difference
of old and new values.
        value -= self.map.get(key, 0)
        self.map[key] = value

        curr = self.root
        for char in key:
            curr = curr.letters.setdefault(char, Trie())
            curr.total += value

    def sum(self, prefix):
```

```
curr = self.root
for char in prefix:
    curr = curr.letters[char]
return curr.total if curr else 0
```
