Epidemic Food

Royalty Free Recipes Re-imagined

Carefully curated professional food library
tailored for your recipes & blogs.

Ismail Elouafiq

May 9, 2017

1 What on Earth is Epidemic Food

Finding the content of foods and recipes online can be tricky. **Epidemic Food** is an imaginary community backed food library managed by nutritionist from around the world. Every recipe is deconstructed into its exact food content which is then deconstructed into the exact nutrient content.

Nutritionists and bloggers around the world are using **Epidemic Food** to give their audience all the knowledge they need for a better health.
Ok, none of that was actually true... :P

The goal of this project is to simply create a food store for the imaginary Epidemic Food App. The food store should enable us to store a list of food and handle queries. This project was implemented in Python (CPython implementation version 2.7.6) implementation. Given that at this stage of the problem additional modules were not necessary, only modules from the standard library have been used to avoid the need of installing new modules. The provided code concerns the final stage of the module. However, the steps taken at each stage are outlined in Section 3. In Section 4 we discuss the case of larger datasets and other possible issues.

2 Structure of the project

The structure of the project is the following:

```
/
├── nomnom
│   ├── __init__.py
│   ├── foodstore.py
│   └── settings.py
└── tests
    └── test_nomnom.py
```

```
    └── test_settings.py
└── Readme.pdf
```

The module *nomnom* contains the food store implementation which is written as a class in the *foodstore.py* file. The *tests* folder contains the tests that were performed in the *test_nomnom.py* file.
The files *settings.py* and *test_settings.py* contain respectively the settings needed for the food store and the settings needed to perform the tests.

3 Implementation Steps

3.1 Stage 1 - Simple Food Storage

On this stage we want to store a list of foods. We are going to use the existing *list* data type in Python to store the food list.
For now, we make the following assumptions:

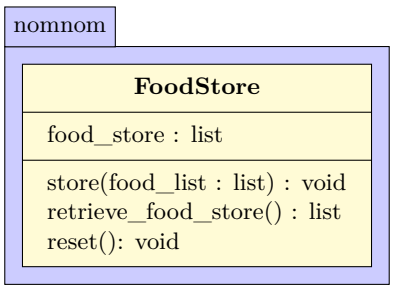
- The elements of the provided list are strings.
- We do not need to store the list on the disk.

3.1.1 Design

With this simple goal in mind, let us design our food store.
We are going to implement a *FoodStore* class where we are going to store the list of food.

- The *food_store* attribute is going to contain the stored list of food.
- The *store* method gets the list of food to be stored as a *list* argument *food_list* and is going to append each element of the list to *food_store*.
- The *retrieve_food_store* method returns the stored food list.
- The *reset* method resets the *food_store* to an empty list.

If the *food_store* can be provided at the instantiation of a *FoodStore* object. Otherwise, it is initialized as an empty string.



3.1.2 Implementation

We start by implementing the *FoodStore* class. At this stage the implementation is straightforward and we take into account the case where an element of the list is not a *string* object. In that case we only ignore it and move on to the next element.
Note: For now, we only ignore non-*string* object entries. It might be better to raise a warning on a later stage.
We end up with the following class:

```
class FoodStore(object):
    """FoodStore class
    Manages Adding and retrieving elements from the food store
    Note: This is the basic case for stage 1
    """
    def __init__(self, food_store = []):
        """Initializes the instance of the FoodStore class

        :param food_store: (optional) initial list of foods in
        the store as a list of strings
        """
        self.food_store = food_store

    def __str__(self):
        """Computes a string representation of the FoodStore

        :return s: string representation
        """
        s = "[>] Store Content\n"
        s += "-----------------\n"
        for food in self.food_store:
            s += "\t"+str(food)+"\n"
        s += "-----------------"
        return s

    def store(self, food_list):
        """Stores the food in the food store

        :param food_list: list of string objects containing
        the food to be added
        """
        for food in food_list:
            # check if food is string
            if isinstance(food, basestring):
                # add food to food store
                self.food_store.append(food)
```

```
    def retrieve_food_store(self):
        """Retrieves the list of food
        :return s: string representation
        """
        return self.food_store
    def reset(self):
        self.food_store = []
```

3.1.3 Testing

The testing was done only using the *unittest* test module from the Python standard library.
The test cases to consider at this stage can be simply:

- **Correctly initializing a food store:** When a food store is initialized correctly no errors should occur.
- **Incorrectly initializing a food store:** When a food store is initialized correctly no errors should occur.
- **Reset:** The reset method should reset the list of food properly.
- **Store:** Given a list of food, the store method should store this list in the food store.
- **Retrieve:** When a the list of food is retrieved it must be identical to the list stored.
- **Store with an outlier:** When a the list of food is stored, if the list contains an element that is not a *string* object, this element is simply ignored and all the rest is properly stored.

For the storage and retrieval tests we want to consider the following examples:

- Food list to be stored: ['banana', 'chocolate cake', 'hot chocolate']
- Food list with an integer outlier: ['banana', 'chocolate cake', 'hot chocolate', 0]

Before we start implementing the tests, we create a Python file, *test_settings.py*, in the *tests* folder to write the test settings. In this file we store the previous examples in a dictionary as follows:

```
test_nomnom_settings = {'food_list':         ['banana', 'chocolate cake',
                                                  'hot chocolate'],
                        'food_list_outlier':['banana', 'chocolate cake',
                                                  'hot chocolate', 0]}
```

We then create the *test_nomnom.py* file to test the previous test cases in the previous example.
At this stage the structure of the project is as follows:

```
/
├── nomnom
│   └── foodstore.py
└── tests
    ├── test_nomnom.py
    └── test_settings.py
```

Hence, to avoid any problems with including the *nomnom* module in the PYTHONPATH for running the tests, we append the path in the beginning of the *test_nomnom.py* file as follows:

```
import sys, os
nomnom_path = os.path.dirname(os.getcwd())
sys.path.append(nomnom_path)
from nomnom import foodstore
```

We then implement a *BasicTest* class to include all the test cases above. The implementation is done as follows:

```
import unittest
from test_settings import test_nomnom_settings as settings

class BasicTest(unittest.TestCase):
    def test_instantiation(self):
        fs = foodstore.FoodStore()

    def test_instantiation_wrong(self):
        with self.assertRaises(TypeError):
            fs = foodstore.FoodStore([], None)

    def test_reset(self):
        fs = foodstore.FoodStore(settings['food_list'])
        fs.reset()
        self.assertEqual(fs.food_store, [])

    def test_storage(self):
        fs = foodstore.FoodStore()
        fs.reset()
        fs.store(settings['food_list'])
        self.assertEqual(fs.food_store, settings['food_list'])

    def test_retrieval(self):
        fs = foodstore.FoodStore()
        fs.reset()
        fs.store(settings['food_list'])
        self.assertEqual(fs.retrieve_food_store(), settings['food_list'])
        self.assertEqual(fs.retrieve_food_store(), fs.food_store)
```

```
    def test_storage_with_outlier(self):
        fs = foodstore.FoodStore()
        fs.reset()
        fs.store(settings['food_list_outlier'])
        self.assertEqual(fs.food_store, settings['food_list'])


if __name__ == '__main__':
    unittest.main()
```

3.2 Stage 2 - Binary Vector Representation

Now we would like to index the documents using their respective binary vector representation.
We need to take the following consideration into account:

- We assume that there are no duplicates in the provided word space
- Some words might not be in the word space. In this case, they will not affect the final binary representation (e.g.: if the word space is ['hot', 'chocolate'] then 'hot chocolate cake' will be represented as '11')
- Different strings might have the same binary representation (e.g: 'hot chocolate', 'chocolate hot' and 'hot chocolate cake' if cake is not in the). We want to keep all of the string documents to be stored and in the same time be able to index them.

3.2.1 Design

Following the previous considerations, we choose to store the food in a dictionary (associative array) where:

- The key will be the index (i.e.: the binary representation of the food document).
- The value will be a list of the food strings. If there is only one element that corresponds to that index then the list will contain one element. If there are two different elements that have the same index, they will be stored in the same list corresponding to their index. If an element is already stored we do not append it again to the list.

Example: For the following word space:

```
['hot', 'cold', 'chocolate', 'milk', 'cake', 'banana', 'orange']
```

And the following list of food:

```
['banana', 'chocolate cake', 'hot chocolate', 'chocolate hot',
'chocolate milk']
```

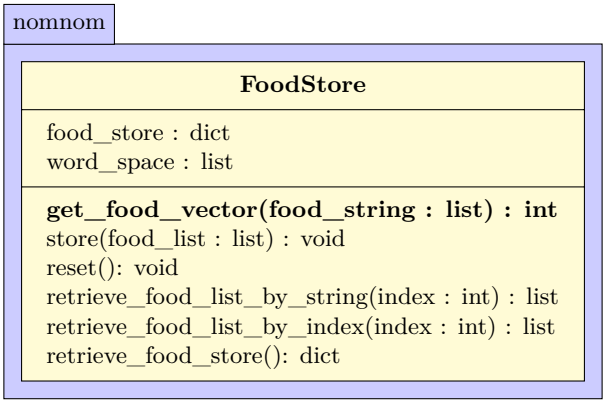
We are going to store the following dictionary:

```
{
  0b10:       ['banana'],
  0b10100:    ['chocolate cake'],
  0b1010000:  ['hot chocolate',
              'chocolate hot'],
  0b11000:    ['chocolate milk']
}
```

Note: We can see the food store as a hash table where the collisions are managed by appending the added element to a list.
The *FoodStore* class will be modified to include the word space at as an attribute *word_space*. If this argument is not given when instantiating a food store it is initialized by default from a default list present in the newly created file *settings.py*. Also, this time the *food_store* attribute will be a dictionary instead of a list.
We also need to implement the following methods:

- The *get_food_vector* method returns the binary representation of a food string. Although the method will return an integer, the integer will be the binary representation itself (e.g.: for the binary representation [1,1,0] the method will return *0b110* which is equal to *6*).
- The *store* method will be modified to store the documents by their respective indexes.
- All previous methods will be modified to take into account the new structure of the food store.

Note: The retrieval functions are only there if we need additional testing without accessing the *food_store* attribute directly.



3.2.2 Implementation

In this section we will only show the implementation of the main methods: *get_food_vector* and *store*.

To store a food document, we are going to go through all the words in the word space. If a word is in the document then we add the digit 1 to the binary representation of the document. Otherwise we just keep 0 as the digit. If a word is present twice in the document it is only counted once (i.e.: 'chocolate chocolate' will have the same binary representation as 'chocolate'). We implement the *get_food_vector* method using some bit manipulation as follows:

```
def get_food_vector(self, food_string):
    """Returns the corresponding vector representation
    of a food string

    :param food_string: string representing the food
    document
    """
    food_list = food_string.strip().split()
    # the binary representation is initialized by zeros
    vector = 0b0
    # we start with the leftmost bit
    shift = self.word_space_length-1
    for element in self.word_space:
        if element in food_list:
                # add the digit 1 in the current position
            vector += 0b1 << shift
        shift -= 1
    return vector
```

We then implement the *store* method by considering the fact that we only store food documents that have not been stored before, and that different documents that have the same index will be stored in the same list. Here is the resulting method:

```
def store(self, food_list):
    """Stores the food in the food store

    :param food_list: list of string objects containing
    the food to be added
    """
    for food in food_list:
        # check if food is string
        if isinstance(food, basestring):
            # add food to food store
            food_vector = self.get_food_vector(food)
            if food_vector not in self.food_store.keys():
                self.food_store[food_vector] = []
            if food not in self.food_store[food_vector]:
                self.food_store[food_vector].append(food)
```

The resulting structure of the project is as follows:

```
/
├── nomnom
│   ├── foodstore.py
│   └── settings.py
└── tests
    ├── test_nomnom.py
    └── test_settings.py
```

Note: For now *settings.py* only contains the default word space but we might need to add other settings to the project later.

3.2.3 Testing

We need to take into account the following test cases:

- **All the previous stage tests:** while considering the new data structure of the food store.
- **Binary representation:** should work as expected for a given word space.
- **Storage:** should take into account the binary representation.
- **Storing a document with a word that is not in the word space:** the word does not affect the binary representation.
- **Storing the same document again:** the document should be stored only once.
- **Storing documents that have the same binary representation:** the documents are stored in the same list under the same key.

These new test cases are handled by adding new tests to the *BasicTest* class.

3.3 Stage 3 - Handling Queries

After the previous stage, the documents are now stored in this format:

```
{
  0b10:       ['banana'],
  0b10100:    ['chocolate cake'],
  0b1010000:  ['hot chocolate',
              'chocolate hot'],
  0b11000:    ['chocolate milk']
}
```

We want to be able to handle queries to the document store. We make the following assumptions:

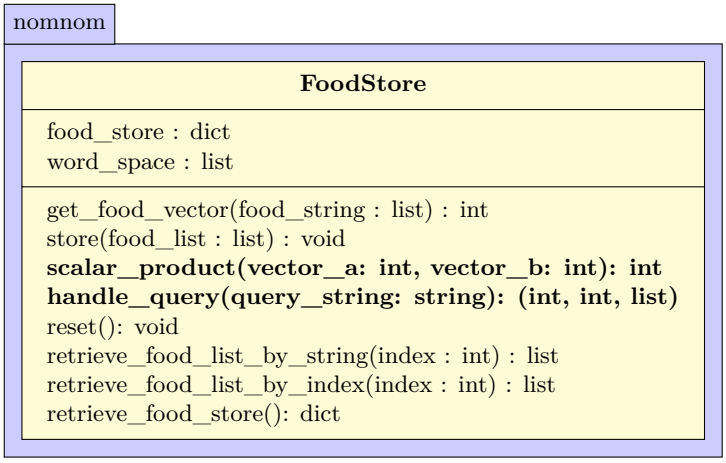
- When responding to a query we will only return the list of food documents corresponding to the index with the highest scalar product.

- We return the score (result of the scalar product), the highest scoring index and the corresponding list of documents.
- If two different indexes have the same score, it does not matter which index is picked as the highest scoring index.

3.3.1 Design

Now, we will mainly need to add two methods:

- A *scalar_product* method which will compute the scalar product of two indexes.
- A *handle_query* method which will take a string object query as a parameter and return the highest scoring document list, the index of the list and the score.



3.3.2 Implementation

To calculate the scalar product we use the *operator* module from the standard library to make use of the *and_* method that performs the bitwise product between two integers. We then compute the number of '1' digits in the binary representation of the result. The method is hence implemented as follows:

```
def scalar_product(self, vector_a, vector_b):
    """Returns the scalar product

    :param vector_a: integer representing the first index
    :param vector_b: integer representing the second index

    :return result: scalar product of the parameters
    Note: This method can be considered static but we
    will keep it this way in case we might change the scalar
```

```
        product by a measure that depends on the food store
        """
        # calculate the bitwise product of the binary representations
        and_vector = operator.and_(vector_a, vector_b)
        # the score is equal to the number of '1' in the binary
        # representation of the and_vector
        result = bin(and_vector).count('1')
        return result
```

The *handle_query* method will receive a query string. If the food store is empty the returned index and list are both *None*. We implement this method by updating the highest scoring document whenever the new score is higher:

```
    def handle_query(self, query_string):
        """Returns the document with the highest scalar product

        :param query_string: string corresponding to the query

        :return highest_score: highest scalar product
        :return highest_scoring_index: vector representation of the document
        :return highest_scoring_document : list of string documents with the
        highest scalar product
        """
        # initialize score and results
        highest_score = 0
        highest_scoring_index = None
        highest_scoring_document = None
        query_vector = self.get_food_vector(query_string)
        for food_vector in self.food_store.keys():
            new_score = self.scalar_product(food_vector, query_vector)
            # update the food document if the score is higher
            if new_score >= highest_score:
                highest_score = new_score
                highest_scoring_index = food_vector
                highest_scoring_document = self.food_store[food_vector]
        return highest_score, highest_scoring_index, highest_scoring_document
```

3.3.3 Testing

Now we will take into account the new following test cases:

- **Calculating the score:** should work properly by returning the correct scalar product of the entry parameters.
- **Handling a query:** should return the result that is expected for the given query. We define queries and their expected results in the *test_settings.py* file.

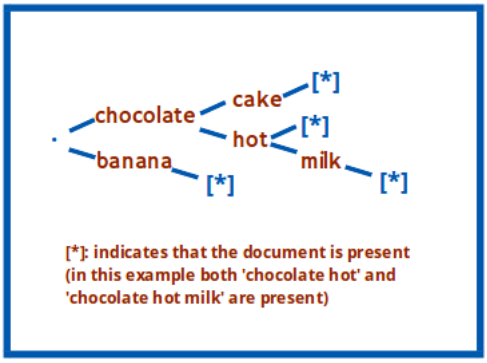


Figure 1: Storage of food using a trie

4 Discussion and Considerations

4.1 Larger datasets

"*How can less be more, it's impossible. More is more.*"
Yngwie Malmsteem

Let us assume that we have a fixed word space and that string documents are limited to a certain number of characters.
The memory in this case scales linearly. Given that N is the number of entries to be stored. Let us see how we can improve this. Here are some possible optimizations:

- We can store the documents in the form of a *trie* (radix tree) instead of a hash table. This example is shown in Figure 4.1. We can do this after sorting the elements of a string document according to some order (for example 'hot chocolate' might become 'chocolate hot').
- To optimize the previous trie we can sort the words in the order of the most commonly used words.
- In the current binary representation there might be some sparse vectors due to the fact that some words are more common than others (word frequency usually follows a distribution similar to a power law distribution) and due to the fact that documents will consist of only a few number of words. Hence, we can encode the strings using *Huffman coding* or similar lossless data compression algorithms.

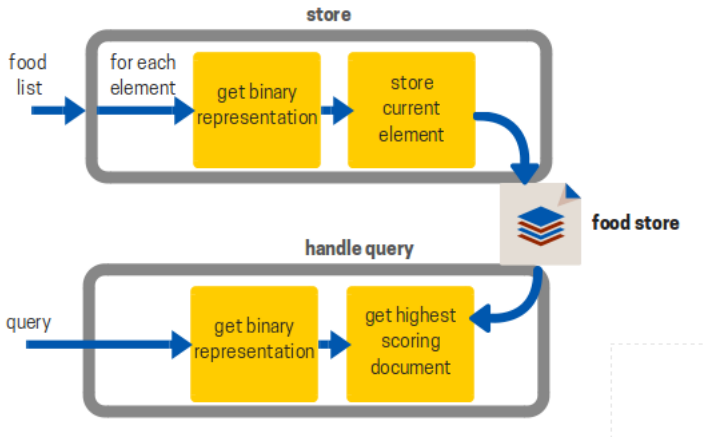


Figure 2: General behavior of the program

4.2 Concurrency issues

The current program works somewhat as shown in Figure 4.2. If we want to use the food store to receive multiple concurrent calls to either store or handle queries, we might need to consider locking both read and write access to the food store. An overly simplified schema would be similar to Figure 4.2.

4.3 User entry issues

If the queries are received directly from a user, these might not correspond to what we assumed. For now, we have ignored the fact that a user might make a mistake in the query.
For example, a user might type 'Chocolate Cqke' instead of 'Chocolate Cake'. In our program, this will just ignore the word 'Cqke'. We might want to return suggestions to the user.

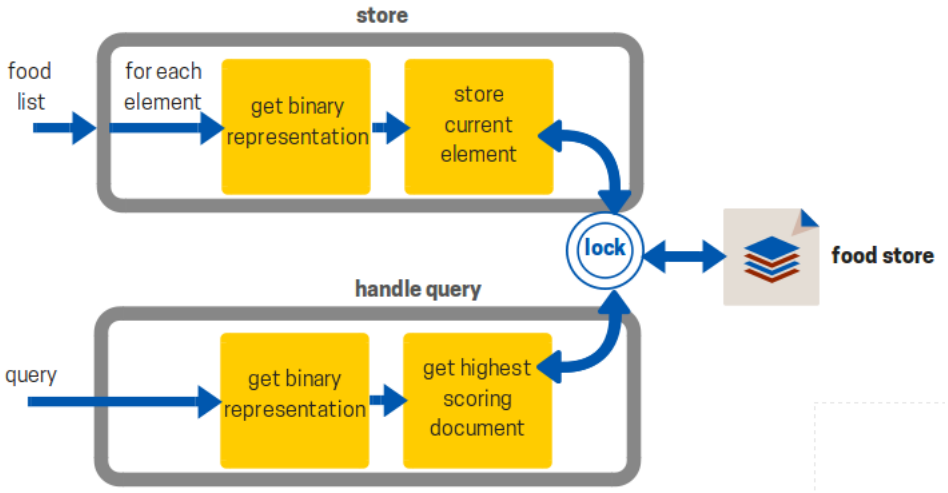


Figure 3: Adding a lock to manage concurrency

Thank you for taking the time to review it. I had fun working on this problem and I hope you'll have as much fun reviewing it. If you have any questions or feedback I would love to hear back from you.

Thanks