Anagrams

The problem

Given a words.txt file containing a newline-delimited list of dictionary words, please implement the Anagrams class so that the get_anagrams() method returns all anagrams from words.txt for a given word.

Bonus requirements:

- Optimise the code for fast retrieval
- Write more tests
- Thread safe implementation

General approach

"An anagram is direct word switch or word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once" (source: wikipedia)

That means that in order to get all the anagrams for a needed word, we don't need to compare the words theirselves but their ordered representation.

Given two words, word1 and word2

If the ordered characters of word1 are the same that the ordered characters of word2,

Then

word1 and word2 are anagrams.

Assumtions

- One given word is anagram of itself.
- Anagrams are **not** case sensitive so "Star" is an anagram of "Tras".
- Special caracters as " ' " are considered as regular caracters too.

Solutions

There is a few options to approach this problem, and this document goes through some of them, from the one which could come first to an inexperienced developer's head to a couple of them with important improvements.

Well see that the first approach, wich implements the trivial solution, has am awful performance, while the second and third one performs thousands of times better with a cost of some extra memory use.

Solution 1

This approach collects all the words in the dictionary and stores them in a list. In order to find the anagrams for a given word, the algorithm needs to sort each of the words in the dictionary to compare them to the sorted given word.

The building of the list is very fast, as no operation involved. However, further searchs are very slow due to the dictionary needs to be complely walked in order to find anagrams.

```
class Anagrams1(Anagrams):
    """
    Very poor performance: This approach collects all the words in the
    dictionary and stores them in a list.
    In order to find the anagrams for a given word, the algorithm needs
    to sort each of the words in the dictionary to compare them to the
    sorted given word.
    """

    def __init__(self, source):
        Anagrams.__init__(self, source)
        self.words = [w[:-2].lower() for w in open(self.source).readlines()]

    @timing
    def get_anagrams(self, word):
        anagrams = []
        word = "".join(c for c in sorted(word.lower()))
        for w in self.words:
            if len(w) != len(word):
                continue
            if "".join(c for c in sorted(w)) == word:
                anagrams.append(w)
        return anagrams

```

Solution 2

In this solution, a python dictionary is created in order to store a pair keys - values, where key is the ordered characters representation of each word in the original dictionary and value is a list containing all the words in the original dictionary where their ordered characters representation is the same that the key.

In this case, collecting the words from the original words dictionary is slightly slower and it requires extra memory (more or less twice, actually) but the performance later on, getting the anagrams for a given word is much better as only indexing the characters ordered representation of the given word will return all its anagrams.

```python
class Anagrams2(Anagrams):
    """
    Much better performance: Create a python dictionary where for each
    original word in the words dictionary, it stores:
         - key: the original sorted word
         - value: all the words that once ordered are the same.
    """

    def __init__(self, source):
        Anagrams.__init__(self, source)
        self.words = {}
        with open(self.source) as words:
            for word in [w[:-2].lower() for w in words]:
                key = "".join(c for c in sorted(word))
                self.words.setdefault(key, [])
                self.words[key].append(word)

    @timing
    def get_anagrams(self, word):
        key = "".join(c for c in sorted(word.lower()))
        return self.words.get(key, [])

```

Solution 3

Similarly to solution 2, buils a python dictionary where the key is the hash of the ordered characters representation for each of the original words and value is a list containing all words where the hash of their ordered characters representation matches the key.

This one should be the best approach in performance and the extra memory used for the keys is fixed to *size of integer* * number of words.

```python
class Anagrams3(Anagrams):
    """
    Hash keys: Create a python dictionary where for each
    original word in the words dictionary, it stores:
      - key: the hash of the original sorted word
      - value: all the words that once ordered have the same hash.
    """

    def __init__(self, source):
        Anagrams.__init__(self, source)
        self.hashes = {}
        with open(self.source) as words:
            for word in [w[:-2].lower() for w in words]:
                key = hash("".join(c for c in sorted(word)))
                self.hashes.setdefault(key, [])
                self.hashes[key].append(word)

    @timing
    def get_anagrams(self, word):
        key = hash("".join(c for c in sorted(word.lower())))
        return self.hashes.get(key, [])

```

Results

Solution 1, as expected, has a very bad performance.

Running each of the aproaches 500 times, Solution 1 is between 5000 and 8000 times slower than Solution 2 and Solution 3

ta/tb	Solution 1	Solution 2	Solution 3
Solution1		7763.218794	7645.291891
Solution2			0.984810

Solution 2 and Solution 3 are almonst the same, being Solution 2 slightly faster than Solution 3 (probably because of the cost of hash).

Solution 3 is, however, less memory consumming.

Figure 1 representes the times for the three solutions.

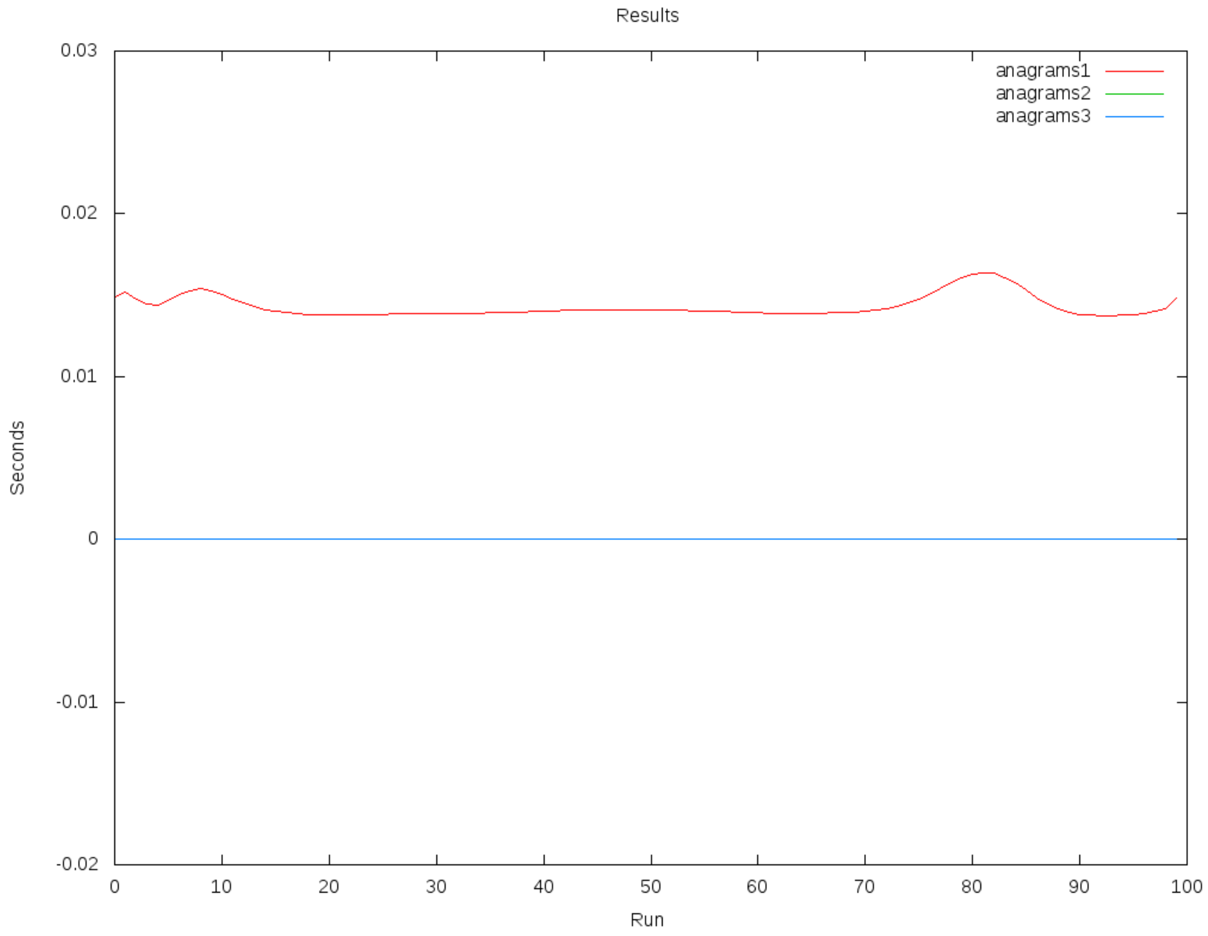

Fig. 1: 50 ran times, solutions 1, 2 and 3

Figure 2 represents times for solutions 2 and 3. Both solutions present a very similar performance.

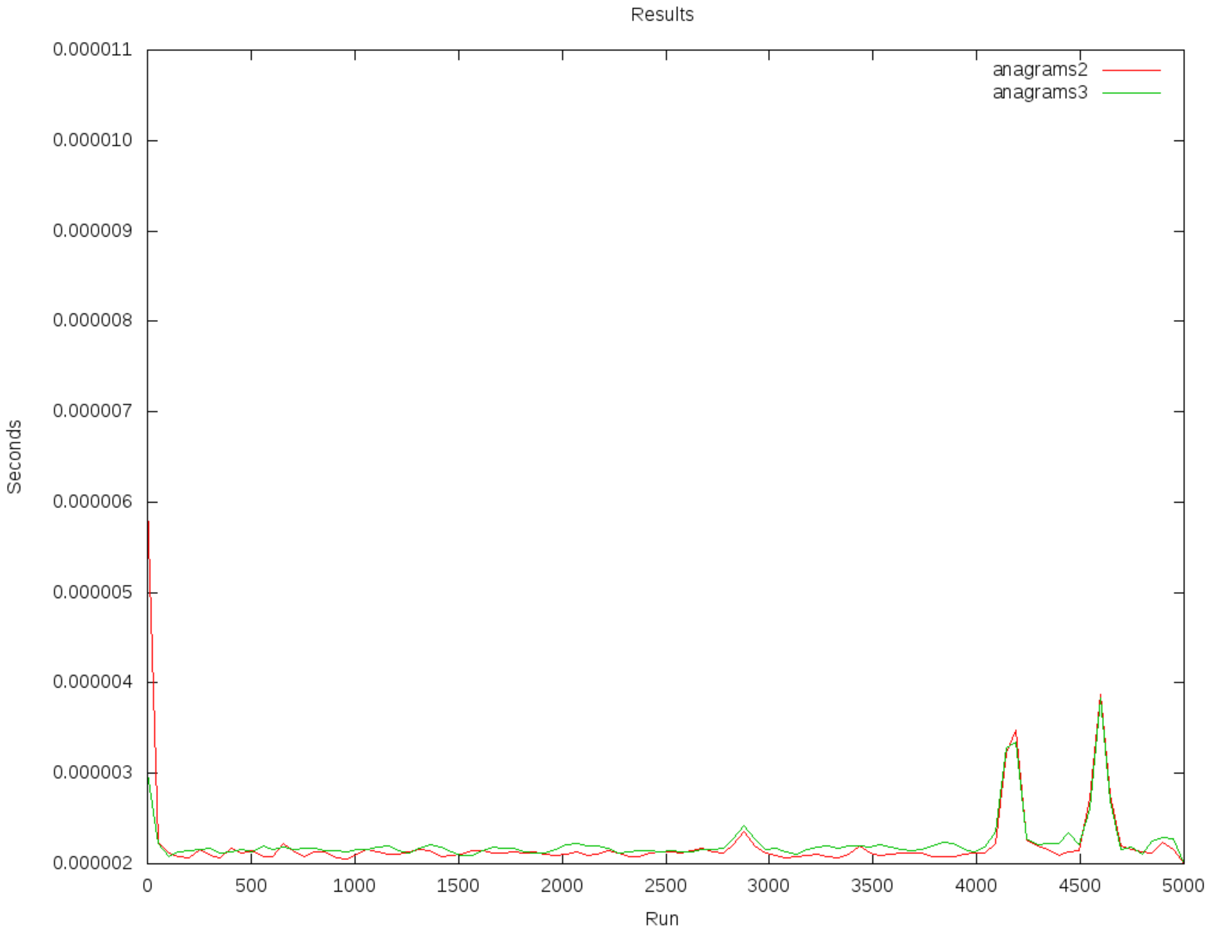

Fig. 2: 5000 ran times, solutions 2 and 3

Having a look to these results, the election of Solution 2 or Solution 3 would depend on which is more important in a real proyect:

- Is it critical to be as fast as possible and to use more memory is not a big deal ?

  Solution 2 wins.
- Is it critical to save memory and having a slighty slower algorithm is suitable ?

  Solution 3 wins.

Latest considerations

- About tests

  An exhaustive test is run covering 100% of the words in the given dictionary
- About threading

  All solutions are thread safe
- About performance

  Solutions 2 and 3 have a very good performance.

Test environment

- Intel(R) Core(TM) i5-5300U CPU @ 2.30GHz.
- Linux Mint 17
- Python 2.7.6

Appendix

Complete code is bellow:

Note: exhaustive tests are massive so they are in a separated file exhaustive.py

```
#!/usr/bin/python

"""
Given a words.txt file containing a newline-delimited list of dictionary
words, please implement the Anagrams class so that the get_anagrams() method
returns all anagrams from words.txt for a given word.

**Bonus requirements:**

  - Optimise the code for fast retrieval
  - Write more tests
  - Thread safe implementation
"""

import os
import time
import unittest


CUR_DIR = os.path.dirname(os.path.realpath(__file__))
OUTPUT_DIR = os.path.join(CUR_DIR, "../output")

class PureVirtualMethod(Exception):
    pass

def timing(f):
    def inner(self, *args, **kwargs):
        t0 = time.time()
        result = f(self, *args, **kwargs)
        t = time.time() - t0
        #print "%s.%s: %0.12f" % (self.__class__.__name__, f.__name__, t)
        return t, result
    return inner

class Statistics(object):
    """
    This class implements different statistics for the different solutions
    It also generates some csv files to be able to process them later on,
    for example with gnuplot
    """
    source = os.path.join(CUR_DIR, 'words.txt')

    def __init__(self):
        self.anagrams1 = Anagrams1(self.source)
        self.anagrams2 = Anagrams2(self.source)
        self.anagrams3 = Anagrams3(self.source)
        self.workers = [self.anagrams1, self.anagrams2, self.anagrams3]

    def ratios(self):
        averages = []
        for worker in self.workers:
            elapsed = 0
            for i in xrange(500):
                t, _ = worker.get_anagrams('plates')
                elapsed += t
            averages.append(elapsed/100.0)

        rat1_2 = averages[0]/averages[1]
        rat1_3 = averages[0]/averages[2]
        rat2_3 = averages[1]/averages[2]

        print "1 vs 2: %f" % rat1_2
        print "1 vs 3: %f" % rat1_3
        print "2 vs 3: %f" % rat2_3

    def gen_csv_all(self):
        output_file = os.path.join(OUTPUT_DIR, "anagrams1.csv")
        output = open(output_file, 'w')
        output.write("anagrams1,anagrams2,anagrams3\n")
        for i in xrange(100):
            t0, _ = self.anagrams1.get_anagrams('plates')
            t1, _ = self.anagrams2.get_anagrams('plates')
            t2, _ = self.anagrams3.get_anagrams('plates')
            output.write("%f, %f, %f\n" %(t0, t1, t2))
        output.close()

    def gen_csv_best(self):
        output_file = os.path.join(OUTPUT_DIR, "anagrams2.csv")
        output = open(output_file, 'w')
        output.write("anagrams2,anagrams3\n")
        for i in xrange(5000):
            t1, _ = self.anagrams2.get_anagrams('plates')
            t2, _ = self.anagrams3.get_anagrams('plates')
            output.write("%f, %f\n" %(t1, t2))
        output.close()


class Anagrams(object):

    def __init__(self, source):
        self.source = source

    def get_anagrams(self, word):
        raise PureVirtualMethod("Pure virtual method. Must be redefined")

# rst-Anagrams1
class Anagrams1(Anagrams):
    """
    Very poor performance: This approach collects all the words in the
    dictionary and stores them in a list.
    In order to find the anagrams for a given word, the algorithm needs
    to sort each of the words in the dictionary to compare them to the
    sorted given word.
    """

    def __init__(self, source):
        Anagrams.__init__(self, source)
        self.words = [w[:-2].lower() for w in open(self.source).readlines()]

    @timing
    def get_anagrams(self, word):
        anagrams = []
        word = "".join(c for c in sorted(word.lower()))
        for w in self.words:
            if len(w) != len(word):
                continue
            if "".join(c for c in sorted(w)) == word:
                anagrams.append(w)
        return anagrams


# rst-Anagrams2
class Anagrams2(Anagrams):
    """
    Much better performance: Create a python dictionary where for each
    original word in the words dictionary, it stores:
         - key: the original sorted word
         - value: all the words that once ordered are the same.
    """

    def __init__(self, source):
        Anagrams.__init__(self, source)
        self.words = {}
        with open(self.source) as words:
            for word in [w[:-2].lower() for w in words]:
                key = "".join(c for c in sorted(word))
                self.words.setdefault(key, [])
                self.words[key].append(word)

    @timing
    def get_anagrams(self, word):
        key = "".join(c for c in sorted(word.lower()))
        return self.words.get(key, [])


# rst-Anagrams3
class Anagrams3(Anagrams):
    """
    Hash keys: Create a python dictionary where for each
    original word in the words dictionary, it stores:
      - key: the hash of the original sorted word
      - value: all the words that once ordered have the same hash.
    """

    def __init__(self, source):
        Anagrams.__init__(self, source)
        self.hashes = {}
        with open(self.source) as words:
            for word in [w[:-2].lower() for w in words]:
                key = hash("".join(c for c in sorted(word)))
                self.hashes.setdefault(key, [])
                self.hashes[key].append(word)
```

```python

    @timing
    def get_anagrams(self, word):
        key = hash("".join(c for c in sorted(word.lower())))
        return self.hashes.get(key, [])

# rst-Tests
class TestAnagrams(unittest.TestCase):

    source = os.path.join(CUR_DIR, 'words.txt')

    def setUp(self):
        self.anagrams1 = Anagrams1(self.source)
        self.anagrams2 = Anagrams2(self.source)
        self.anagrams3 = Anagrams3(self.source)

    def test_no_word(self):
        _, r =  self.anagrams1.get_anagrams("")
        self.assertEqual([], r)
        _, r =  self.anagrams2.get_anagrams("")
        self.assertEqual([], r)
        _, r =  self.anagrams3.get_anagrams("")
        self.assertEqual([], r)

    def test_pure_virtual(self):
        class AnagramsX(Anagrams):
            def __init__(self, source):
                Anagrams.__init__(self, source)

        anagrams = AnagramsX(self.source)
        self.assertRaises(PureVirtualMethod, anagrams.get_anagrams, 'pastel')

    def test_exaustive(self):
        """
        This tests tests all anagrams for all words in the given
        dictionary !!
        They are not ran as individual tests because the required resources
        for it would be massive.
        """
        import exhaustive
        exhaustive.test_exhaustive(self)


if __name__ == '__main__':
    unittest.main()
    #statistics = Statistics()
    #statistics.ratios()
    #statistics.gen_csv_all()
    #statistics.gen_csv_best()
```