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 given word, we don't need to compare the words their selves 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.

Assumptions

- One given word is anagram of itself.
- Anagrams are **not** case sensitive so "Star" is an anagram of "Rats".
- Special characters as " ' " are considered as regular characters 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, which implements the trivial solution, has an awful performance, while the second performs thousands of times better.

Solution 1: Brute force

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 searches are very slow due to the dictionary needs to be completely 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: sorted characters keys

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.

```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, [])

```

Results

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

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

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

Solution 2 performance shows much better performance, due to the fact that searching in a Python dictionary (where keys are hashes) is very efficient.

Figure 1 represents the times for the two solutions.

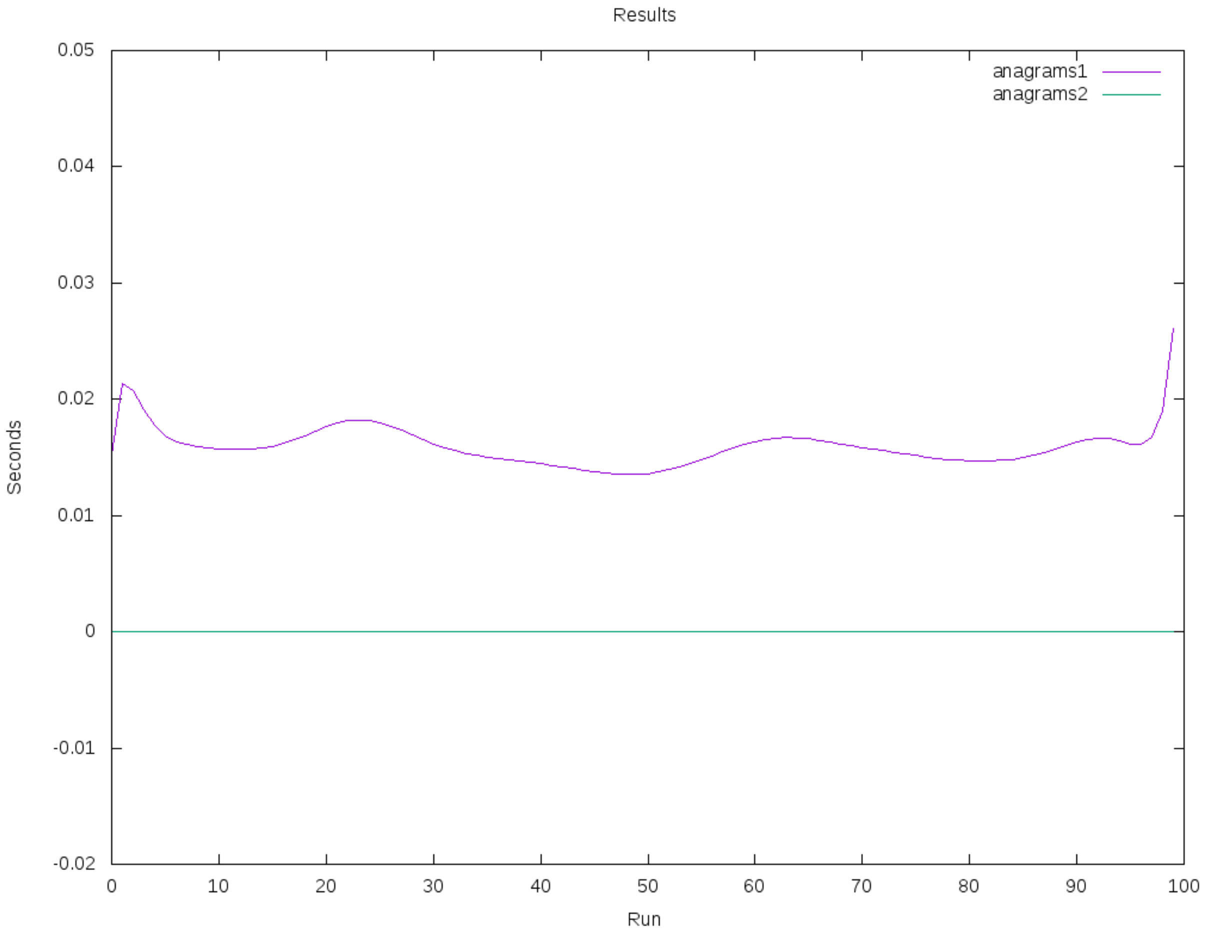

Fig. 1: 100 run times, solutions 1 and 2

Figure 2 represents times for solution 2.

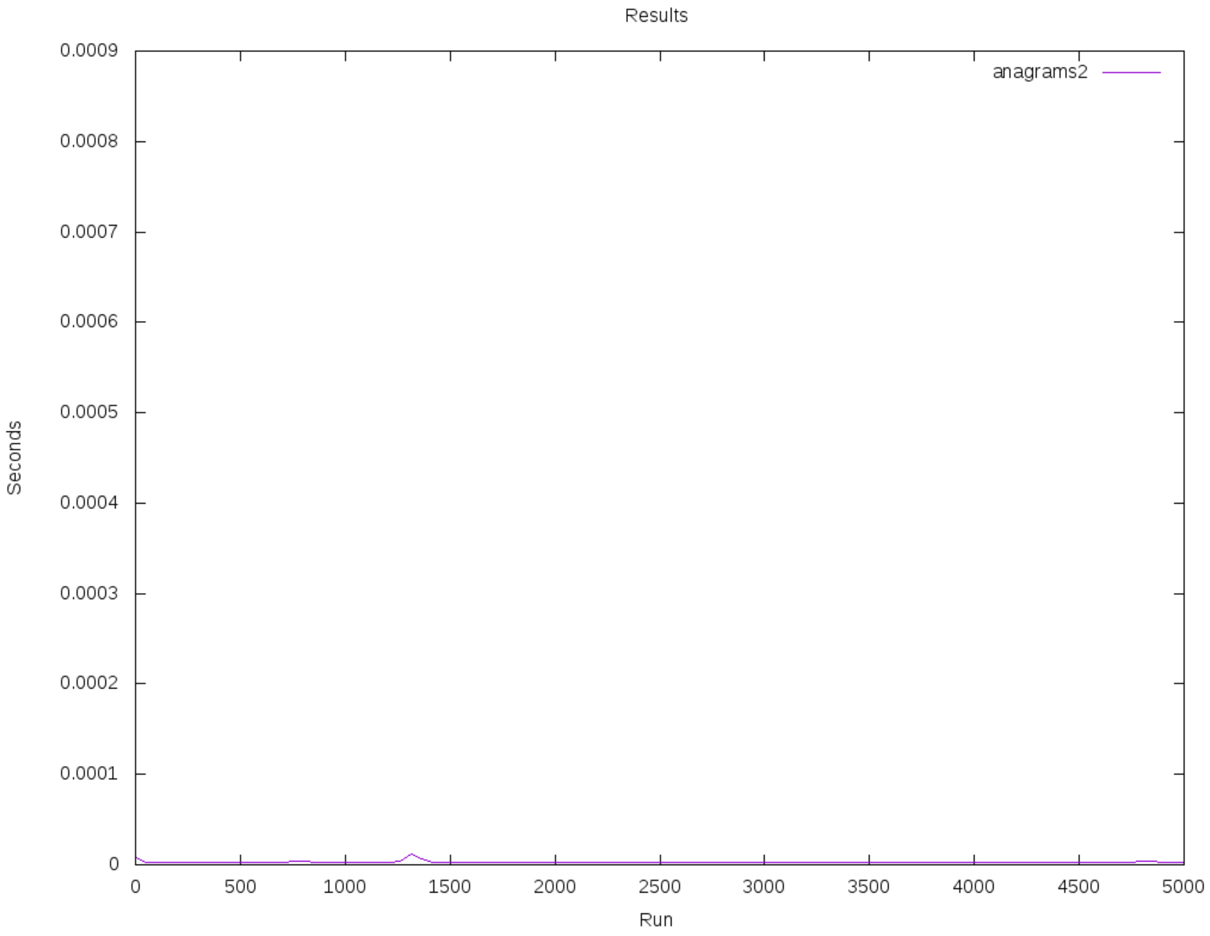

Fig. 2: 5000 ran times, solution 2

Latest considerations

- About tests

  Exhaustive tests are running against every single word in the provide dictionary
- About threading

  All solutions are thread safe
- About performance

  Solutions 2 has 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:

```python
#!/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


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.workers = [self.anagrams1, self.anagrams2]

    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]

        print "1 vs 2: %f" % rat1_2

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

    def gen_csv_best(self):
        output_file = os.path.join(OUTPUT_DIR, "anagrams2.csv")
        output = open(output_file, 'w')
        output.write("anagrams2\n")
        for i in xrange(5000):
            t1, _ = self.anagrams2.get_anagrams('plates')
            output.write("%f\n" %(t1))
        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, [])
# end-rst-Anagrams2

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