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 am awful performance, while the second and third one performs thousands of times better with a cost of some extra memory use.

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.

```python
 96 class Anagrams1(Anagrams):
 97     """
 98     Very poor performance: This approach collects all the words in the
 99     dictionary and stores them in a list.
100     In order to find the anagrams for a given word, the algorithm needs
101     to sort each of the words in the dictionary to compare them to the
102     sorted given word.
103     """
104
105     def __init__(self, source):
106         Anagrams.__init__(self, source)
107         self.words = [w[:-2].lower() for w in open(self.source).readlines()]
108
109     @timing
110     def get_anagrams(self, word):
111         anagrams = []
112         word = "".join(c for c in sorted(word.lower()))
113         for w in self.words:
114             if len(w) != len(word):
115                 continue
116             if "".join(c for c in sorted(w)) == word:
117                 anagrams.append(w)
118         return anagrams
119
```

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.

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
122  class Anagrams2(Anagrams):
123      """
124      Much better performance: Create a python dictionary where for each
125      original word in the words dictionary, it stores:
126          - key: the original sorted word
127          - value: all the words that once ordered are the same.
128      """
129
130      def __init__(self, source):
131          Anagrams.__init__(self, source)
132          self.words = {}
133          with open(self.source) as words:
134              for word in [w[:-2].lower() for w in words]:
135                  key = "".join(c for c in sorted(word))
136                  self.words.setdefault(key, [])
137                  self.words[key].append(word)
138
139      @timing
140      def get_anagrams(self, word):
141          key = "".join(c for c in sorted(word.lower()))
142          return self.words.get(key, [])
143
```

Solution 3: hash keys

Similarly to solution 2, builds 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
146 class Anagrams3(Anagrams):
147     """
148     Hash keys: Create a python dictionary where for each
149     original word in the words dictionary, it stores:
150       - key: the hash of the original sorted word
151       - value: all the words that once ordered have the same hash.
152     """
153
154     def __init__(self, source):
155         Anagrams.__init__(self, source)
156         self.hashes = {}
157         with open(self.source) as words:
158             for word in [w[:-2].lower() for w in words]:
159                 key = hash("".join(c for c in sorted(word)))
160                 self.hashes.setdefault(key, [])
161                 self.hashes[key].append(word)
162
163     @timing
164     def get_anagrams(self, word):
165         key = hash("".join(c for c in sorted(word.lower())))
166         return self.hashes.get(key, [])
167
```

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	Solution 3
Solution1		7763.218794	7645.291891
Solution2			0.984810

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

Solution 3 is, however, less memory consuming.

Figure 1 represents the times for the three solutions.
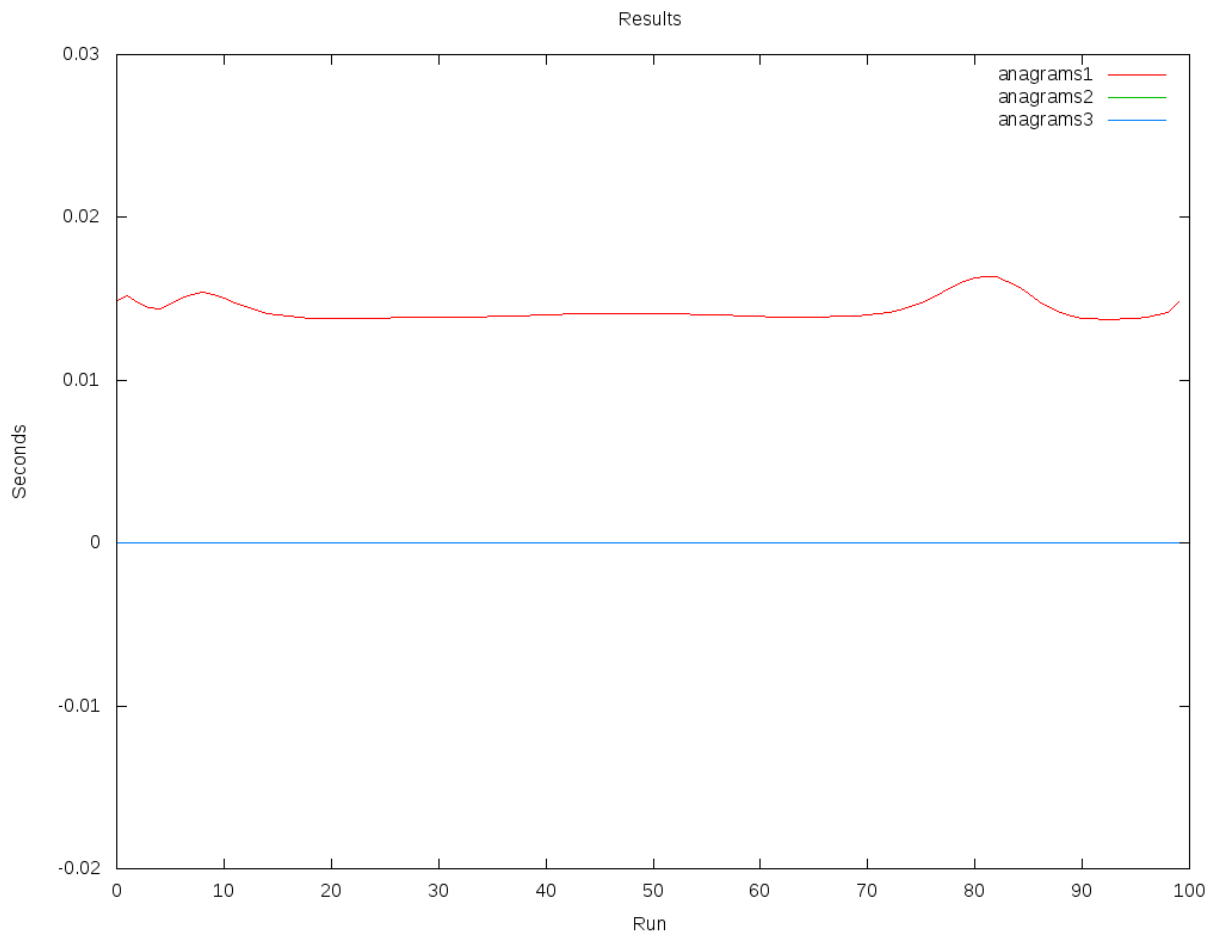

Fig. 1: 100 run times, solutions 1, 2 and 3

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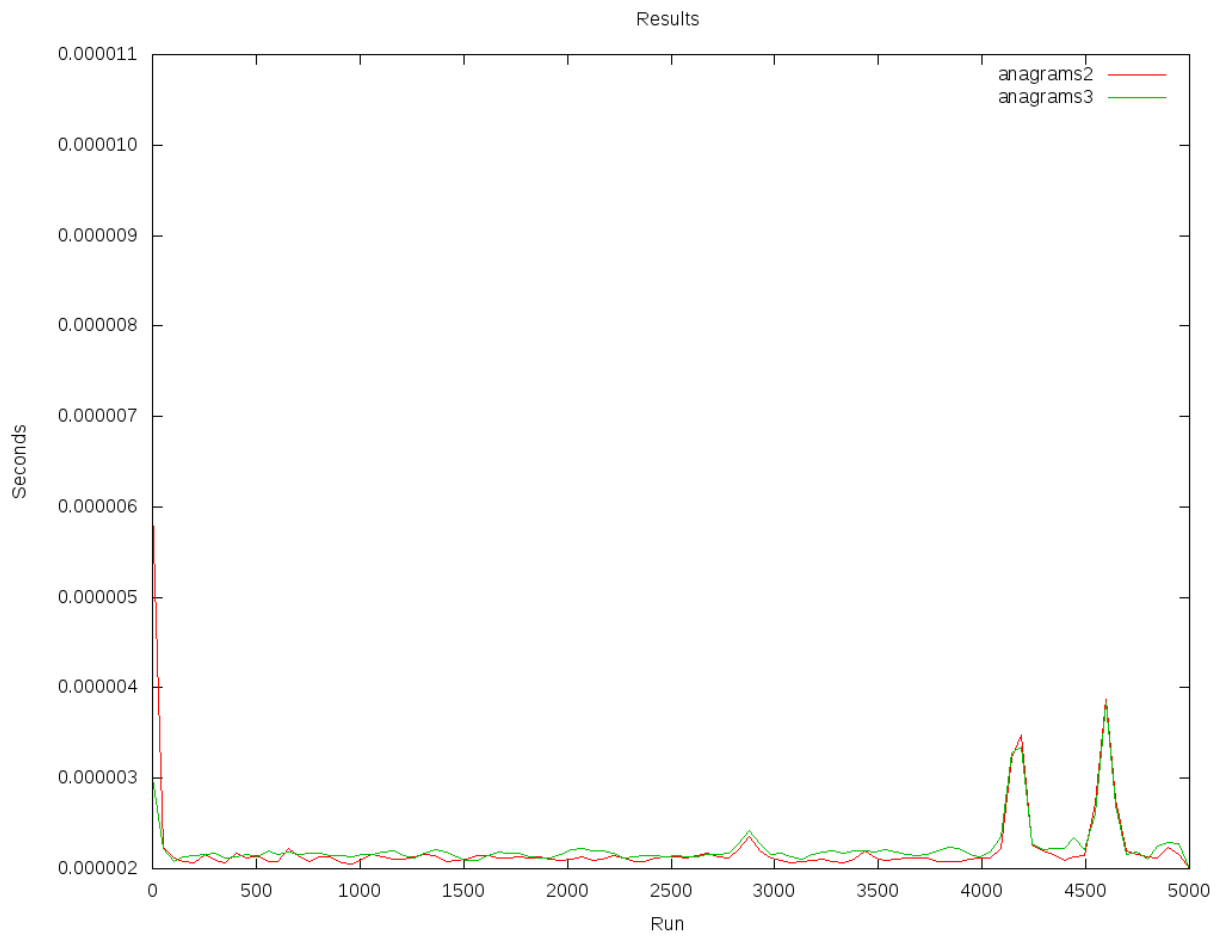

Fig. 2: 5000 ran times, solutions 2 and 3

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

- 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 slightly slower algorithm is suitable ?

  Solution 3 wins.

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 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:

```python
1 #!/usr/bin/python
2
3 """
4 Given a words.txt file containing a newline-delimited list of dictionary
5 words, please implement the Anagrams class so that the get_anagrams() method
6 returns all anagrams from words.txt for a given word.
7
8 **Bonus requirements:**
9
10    - Optimise the code for fast retrieval
11    - Write more tests
12    - Thread safe implementation
13 """
14
15 import os
16 import time
17
18
19 CUR_DIR = os.path.dirname(os.path.realpath(__file__))
20 OUTPUT_DIR = os.path.join(CUR_DIR, "../output")
21
22 class PureVirtualMethod(Exception):
23     pass
24
25 def timing(f):
26     def inner(self, *args, **kwargs):
27         t0 = time.time()
28         result = f(self, *args, **kwargs)
29         t = time.time() - t0
30         #print "%s.%s: %0.12f" % (self.__class__.__name__, f.__name__, t)
31         return t, result
32     return inner
33
34 class Statistics(object):
35     """
36     This class implements different statistics for the different solutions
37     It also generates some csv files to be able to process them later on,
38     for example with gnuplot
39     """
40     source = os.path.join(CUR_DIR, 'words.txt')
41
42     def __init__(self):
43         self.anagrams1 = Anagrams1(self.source)
44         self.anagrams2 = Anagrams2(self.source)
45         self.anagrams3 = Anagrams3(self.source)
46         self.workers = [self.anagrams1, self.anagrams2, self.anagrams3]
47
48     def ratios(self):
49         averages = []
50         for worker in self.workers:
51             elapsed = 0
52             for i in xrange(500):
```

```python
53                   t, _ = worker.get_anagrams('plates')
54                   elapsed += t
55               averages.append(elapsed/100.0)
56
57           rat1_2 = averages[0]/averages[1]
58           rat1_3 = averages[0]/averages[2]
59           rat2_3 = averages[1]/averages[2]
60
61           print "1 vs 2: %f" % rat1_2
62           print "1 vs 3: %f" % rat1_3
63           print "2 vs 3: %f" % rat2_3
64
65       def gen_csv_all(self):
66           output_file = os.path.join(OUTPUT_DIR, "anagrams1.csv")
67           output = open(output_file, 'w')
68           output.write("anagrams1,anagrams2,anagrams3\n")
69           for i in xrange(100):
70               t0, _ = self.anagrams1.get_anagrams('plates')
71               t1, _ = self.anagrams2.get_anagrams('plates')
72               t2, _ = self.anagrams3.get_anagrams('plates')
73               output.write("%f, %f, %f\n" %(t0, t1, t2))
74           output.close()
75
76       def gen_csv_best(self):
77           output_file = os.path.join(OUTPUT_DIR, "anagrams2.csv")
78           output = open(output_file, 'w')
79           output.write("anagrams2,anagrams3\n")
80           for i in xrange(5000):
81               t1, _ = self.anagrams2.get_anagrams('plates')
82               t2, _ = self.anagrams3.get_anagrams('plates')
83               output.write("%f, %f\n" %(t1, t2))
84           output.close()
85
86
87 class Anagrams(object):
88
89     def __init__(self, source):
90         self.source = source
91
92     def get_anagrams(self, word):
93         raise PureVirtualMethod("Pure virtual method. Must be redefined")
94
95 # rst-Anagrams1
96 class Anagrams1(Anagrams):
97     """
98     Very poor performance: This approach collects all the words in the
99     dictionary and stores them in a list.
100    In order to find the anagrams for a given word, the algorithm needs
101    to sort each of the words in the dictionary to compare them to the
102    sorted given word.
103    """
104
105    def __init__(self, source):
106        Anagrams.__init__(self, source)
107        self.words = [w[:-2].lower() for w in open(self.source).readlines()]
108
```

```python
109         @timing
110         def get_anagrams(self, word):
111             anagrams = []
112             word = "".join(c for c in sorted(word.lower()))
113             for w in self.words:
114                 if len(w) != len(word):
115                     continue
116                 if "".join(c for c in sorted(w)) == word:
117                     anagrams.append(w)
118             return anagrams
119
120
121 # rst-Anagrams2
122 class Anagrams2(Anagrams):
123     """
124     Much better performance: Create a python dictionary where for each
125     original word in the words dictionary, it stores:
126         - key: the original sorted word
127         - value: all the words that once ordered are the same.
128     """
129
130     def __init__(self, source):
131         Anagrams.__init__(self, source)
132         self.words = {}
133         with open(self.source) as words:
134             for word in [w[:-2].lower() for w in words]:
135                 key = "".join(c for c in sorted(word))
136                 self.words.setdefault(key, [])
137                 self.words[key].append(word)
138
139     @timing
140     def get_anagrams(self, word):
141         key = "".join(c for c in sorted(word.lower()))
142         return self.words.get(key, [])
143
144
145 # rst-Anagrams3
146 class Anagrams3(Anagrams):
147     """
148     Hash keys: Create a python dictionary where for each
149     original word in the words dictionary, it stores:
150       - key: the hash of the original sorted word
151       - value: all the words that once ordered have the same hash.
152     """
153
154     def __init__(self, source):
155         Anagrams.__init__(self, source)
156         self.hashes = {}
157         with open(self.source) as words:
158             for word in [w[:-2].lower() for w in words]:
159                 key = hash("".join(c for c in sorted(word)))
160                 self.hashes.setdefault(key, [])
161                 self.hashes[key].append(word)
162
163     @timing
164     def get_anagrams(self, word):
```

```
165         key = hash("".join(c for c in sorted(word.lower())))
166         return self.hashes.get(key, [])
167
168 # rst-main
169 if __name__ == '__main__':
170     pass
171     #statistics = Statistics()
172     #statistics.ratios()
173     #statistics.gen_csv_all()
174     #statistics.gen_csv_best()
```