

# 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.

We'll see that the first approach, which implements the trivial solution, has an 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.

```
89 class Anagrams1(Anagrams):
90     """
91     Hi Ovidiu
92     Very poor performance: This approach collects all the words in the
93     dictionary and stores them in a list.
94     In order to find the anagrams for a given word, the algorithm needs
95     to sort each of the words in the dictionary to compare them to the
96     sorted given word.
97     """
98
99     def __init__(self, source):
100         Anagrams.__init__(self, source)
101         self.words = [w[:-2].lower() for w in open(self.source).readlines()]
102
103     @timing
104     def get_anagrams(self, word):
105         anagrams = []
106         word = "".join(c for c in sorted(word.lower()))
107         for w in self.words:
108             if len(w) != len(word):
109                 continue
110             if "".join(c for c in sorted(w)) == word:
111                 anagrams.append(w)
112         return anagrams
113
```

## 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.

```
116 class Anagrams2(Anagrams):
117     """
118     Much better performance: Create a python dictionary where for each
119     original word in the words dictionary, it stores:
120         - key: the original sorted word
121         - value: all the words that once ordered are the same.
122     """
123
124     def __init__(self, source):
125         Anagrams.__init__(self, source)
126         self.words = {}
127         with open(self.source) as words:
128             for word in [w[:-2].lower() for w in words]:
129                 key = "".join(c for c in sorted(word))
130                 self.words.setdefault(key, [])
131                 self.words[key].append(word)
132
133     @timing
134     def get_anagrams(self, word):
135         key = "".join(c for c in sorted(word.lower()))
136         return self.words.get(key, [])
137
```

## Results

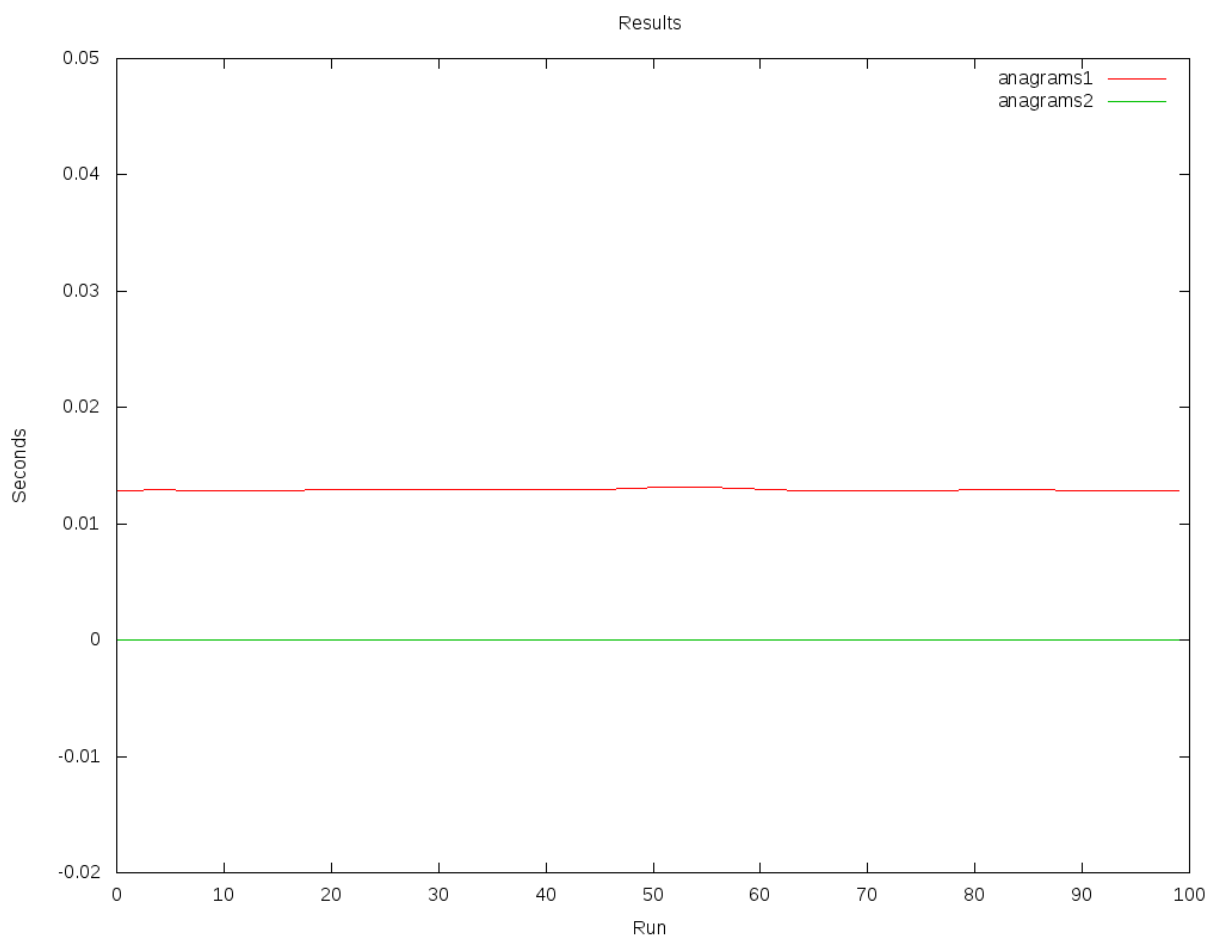
Solution 1, as expected, has a very bad performance.

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

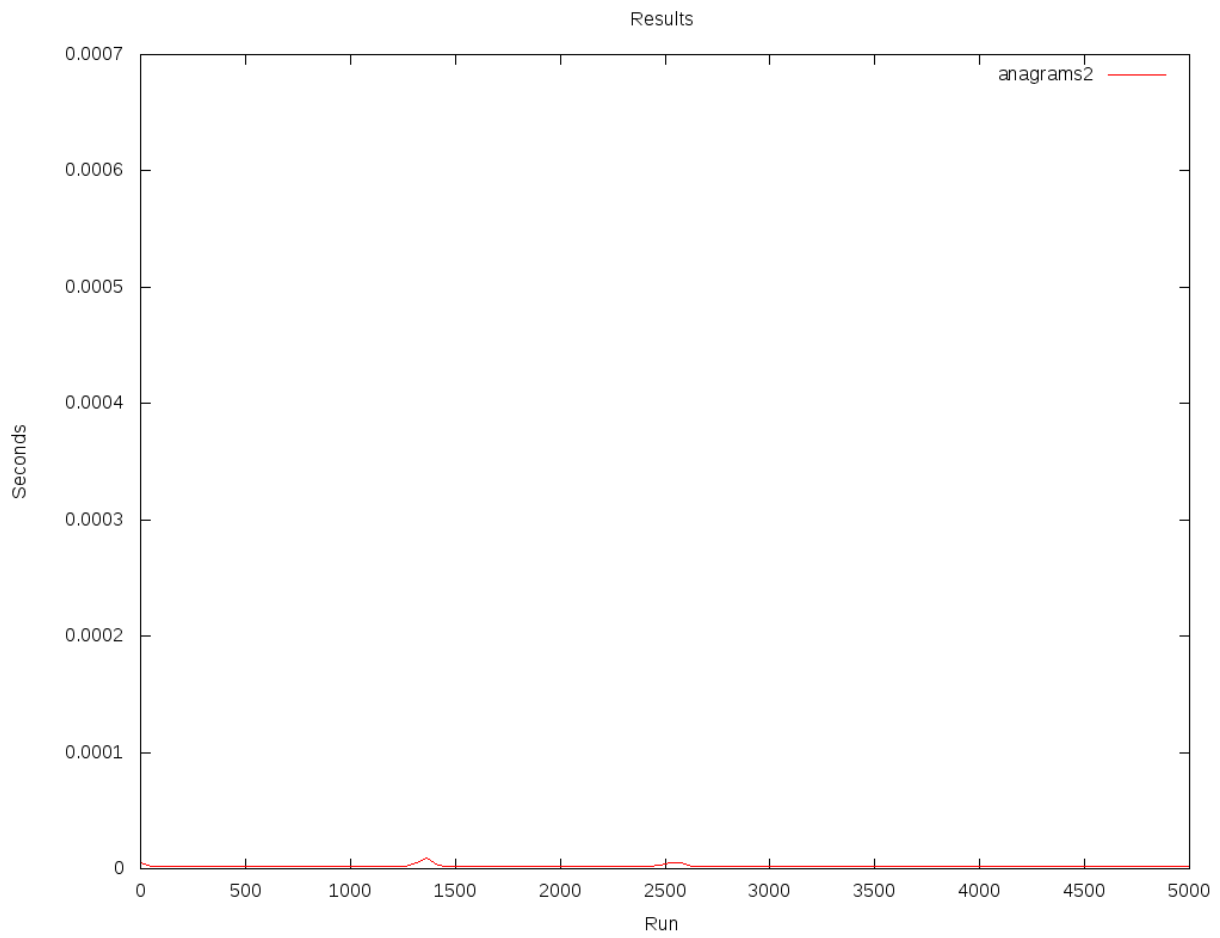
Solution 2 performance shows much better performance, due to the fact that searching in a Python dictionary ( where keys are hashes ) is very efficient.

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:

```
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.workers = [self.anagrams1, self.anagrams2]
46
47     def ratios(self):
48         averages = []
49         for worker in self.workers:
50             elapsed = 0
51             for i in xrange(500):
52                 t, _ = worker.get_anagrams('plates')
```

```

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

```



```

109         continue
110         if "".join(c for c in sorted(w)) == word:
111             anagrams.append(w)
112     return anagrams
113
114
115 # rst-Anagrams2
116 class Anagrams2(Anagrams):
117     """
118     Much better performance: Create a python dictionary where for each
119     original word in the words dictionary, it stores:
120         - key: the original sorted word
121         - value: all the words that once ordered are the same.
122     """
123
124     def __init__(self, source):
125         Anagrams.__init__(self, source)
126         self.words = {}
127         with open(self.source) as words:
128             for word in [w[:-2].lower() for w in words]:
129                 key = "".join(c for c in sorted(word))
130                 self.words.setdefault(key, [])
131                 self.words[key].append(word)
132
133     @timing
134     def get_anagrams(self, word):
135         key = "".join(c for c in sorted(word.lower()))
136         return self.words.get(key, [])
137 # end-rst-Anagrams2
138
139 # rst-main
140 if __name__ == '__main__':
141     statistics = Statistics()
142     statistics.ratios()
143     statistics.gen_csv_all()
144     statistics.gen_csv_best()

```