Test Case Study 3 Analysis and Write Up

# Question #1:

Run a performance test that does 2M searches with random search terms, and measures execution time. Which approach is fastest? Why?

**Answer:**

The regex approach was fastest. The simple string match was the second fastest. The indexed approach came in last. My best guess as to why the indexed approach, which I assumed would come in first, came in last is that other pieces of the implementation subsumed what should have been a simple, speed dictionary look-up. The Python folks have coded the regex to be very speedy, so it is understandable that it bested the simple string match.

## Further thoughts:

I did not realize until I embarked on this project that the timing analysis would be the hard part. I did not realize how finnicky timing of code can be. I started, initially, with a naïve timing implementation. You can see this attempt under the performance\_test/performance\_test\_naive.py. It was only after reading Tim Peters’ Introduction to Chapter 17 Algorithms in the Python Cookbook (<https://www.oreilly.com/library/view/python-cookbook/0596001673/ch17.html>) that I realized what I was in for.

## Data from the performance\_test\_naive.py

After 2 million searches, the trial times were as follows

|  |  |
| --- | --- |
| **Search Approach** | Time (Hours:Minutes:Seconds) |
| Regex matching | 0:36:41.032512 |
| Simple string matching | 0:44:21.763456 |
| Index without md5sum checking | **0:56:30.327846** |
| Index with md5sum checking (See addendum) | 1:02:15.441282 |

The regex solution relies on the ‘re’ built-in Python library. I assume this is coded to be fast, and it clocks in better than both the simple string approach and the indexed approach.

The simple string-matching approach was the brute force method, but it still bested my implementation of the indexed approach.

The indexed approach front-end loads the work by indexing the entire file and performs the indexing only once. My solution cached this index into a file, so the work did not have to be repeated. I thought this would have been labor saving and time saving. I wanted to make this search script a command line tool that was invoked repeatedly rather than a tool that hung around with a command prompt. Therefore, I had to cache the index in a file that needed to be reloaded rather than something that stayed in working memory.

**Addendum:**

As a safety precaution, for the indexed method, I thought I should do an md5sum calculation on the text file that I was searching to ensure that the file had not changed between the time that I indexed the file and the time I was searching it. This proved to be a naïve and not advantageous to performanc decision as it made the index method come in last out of all three methods with a time of 1:02:15.441282.

## Data from performance\_test.py

Per what I read in the Python Cookbook, I rewrote the different search algorithms to have timing components built into them to try and eliminate some of the ‘chaff’ from my earlier timing analysis. I shut down all other running applications on my laptop and even disconnected my network cable.

Here are the results. This time around, I ran the 2 million plus iterations and calculated average times for them.

|  |  |
| --- | --- |
| **Search Approach** | **Average Time (Seconds)** |
| Regex matching | 0.000051593464733 |
| Simple string matching | 0.0002325254755401531 |
| Index without md5sum checking | 0.0005308850794731124 |
| Index with md5sum checking (See addendum) | Didn’t run this one. |

**Analysis**:

As before, in this *hopefully* better diagnostic attempt the regex search comes in first. It is 4.5 times faster than the simple string search. It is 10.3 times faster than my index matching approach. The index match again comes in dead last, subsumed by other steps in the implementation.

# Question #2:

Provide some thoughts on what you would do on the software or hardware side to make this program scale to handle massive content and/or very large request volume (5000 requests/second or more).

**Answers:**

**Software:**

I would have the requests load balanced across the hardware, so that each piece of hardware is taking its fair share. If this was some daemon listening on a socket, I would have it spinning off requests to be handled by separate processes. Basically, I would try to do as much in parallel (both hardware parallelization and software parallelization) as possible.

**Hardware:**

I would ensure that I had enough hardware and beefy enough hardware. You would want machines with enough working memory (RAM) and multiple cores. If you do not want to spend money on beefier machines, you can supplement with more less beefy machines and simply ask less of them.

I would want to ensure that my network and switches could handle the expected bandwidth and not bog down.