Regex Performance

Content Prepared By: Chandra Lingam, Cotton Cola Designs LLC

Regex Performance

Type of patterns that can cause performance issues

Fix for performance issues

Module Versus Compiled Regex Method Invocation

Backtracking Exponential Delay Example

- Problem: Match a word
- Pattern: ^(\w*)*$
- Text (Positive): 12345678901234567890
- Text (partial match): 12345678901234567890!

Quantifier for the word character (0 or more)

Group has a quantifier which indicates 0 or more groups

- Pattern works perfectly for positive matches. With partial matches, performance degrades rapidly and every additional character doubles the response time.

Issue

Pattern: ^(\w*)*$ Text:12345!

Match 1: 1
Match 2: 12
Match 3: 123
Match 4: 1234
Match 5: 12345 - ! Does not match \w. \w* stops now. And end of string $ match fails.

Issues and Fix

- Pattern: (\w*)* => (\w*)(\w*)(\w*)….
- Multiple similar greedy patterns capturing same characters can cause serious performance issues
- Option 1: No need to have a group level quantifier

  ^(\w*)*$ => ^(\w*)$
- Option 2: Precise terminating condition. Every word in group should end in a word boundary.

  ^(\w*)*$ => ^(\w*\b)*$
- Disable group capture

  ^(\w*)$ => ^\w*$ or ^(?:\w*)$

Compiled Regex Objects vs Module Methods

You have two ways to invoke regular expression functionality

- Create a compiled object and invoke methods of that object
- Use re module methods directly

Compiled object method provides additional fine tuning parameters. For example: start position, ending position of search and so forth.

re module caches the compiled version of patterns and reuses them

- Python 3 Pattern Cache Size is 512
- Python 2 Pattern Cache size is 100
- When patterns exceed cache size, current implementation of re simply clears the entire cache.

Best Practice: For high performance and/or high frequency invocation of patterns – Use compiled objects, hold reference to the objects and reuse them.