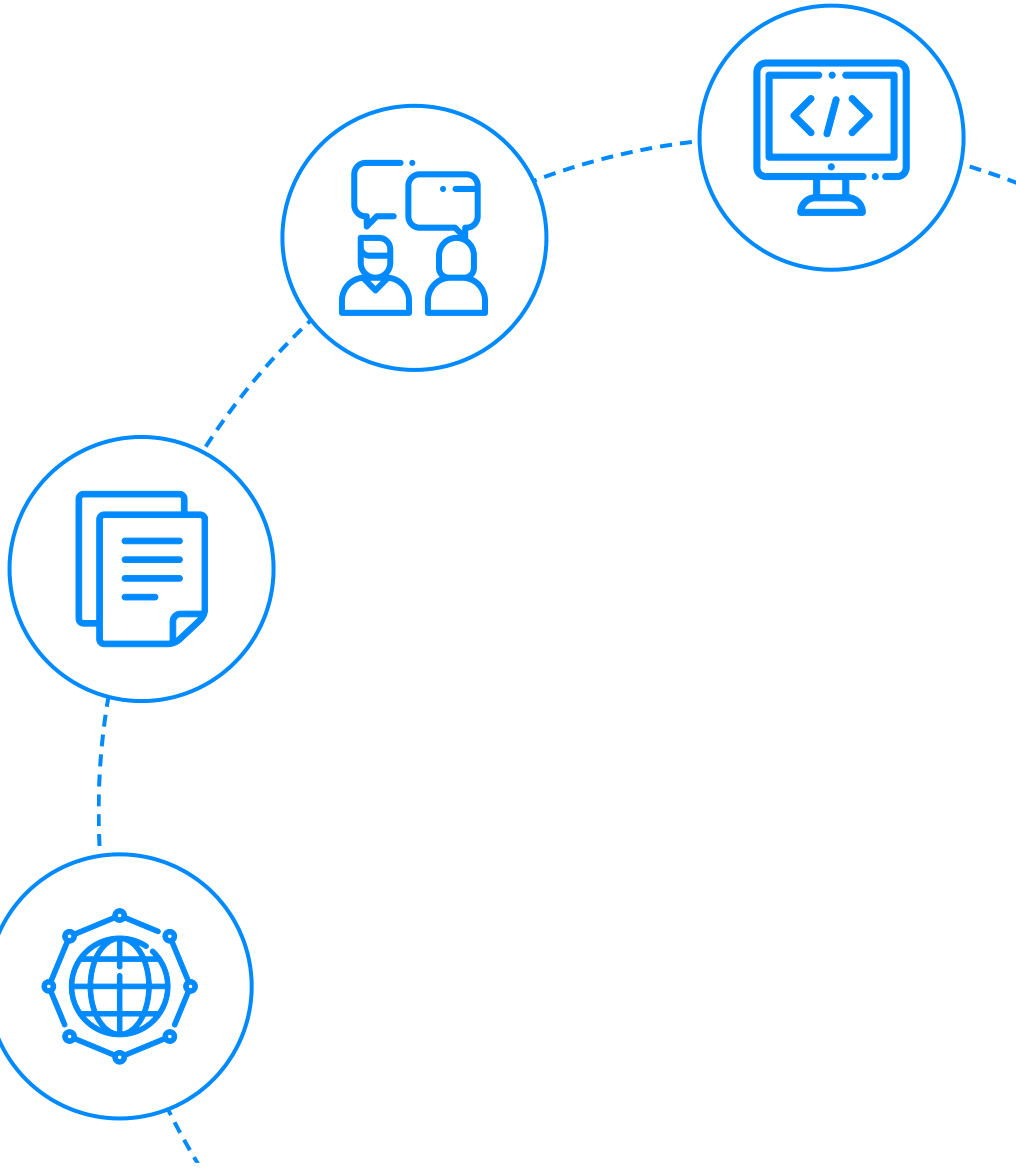
InterviewBit

Regex Cheat Sheet



To view the live version of the page, click here.

Contents

Basic Regex Commands

Advanced Regex Commands

Possible Performance pitfalls

Tips to increase the performance

Let's get Started

Introduction

A regular expression (regex in short) is a pattern in input text that the regular expression engine tries to match. One or more character literals, operators, or structures make up a pattern. It's particularly good at searching for and manipulating text strings, as well as processing text files. A single regex line can easily replace dozens of lines of programming code. In this blog, you will come across the basic to advanced regex concepts that would help you in your development task.

Why Regex?

Searching and replacement operations are made easy by regular expressions. Finding a substring that matches a pattern and replacing it with something else is a common use case. Regex is renowned for the IT expertise that dramatically boosts productivity in all computer tasks.

Basic Regex Commands

1. Characters

The character class is the most basic regex concept. It converts a small group of characters to match a larger number of characters.

Character	Description
\	Escape character
.	Any character
\s	whitespace
\S	Not white space
\d	Digit
\D	Not digit
\w	Word character
\W	Not word character
\b	Word boundary
\B	Not word boundary
^	Beginning of string
$	End of string

For example:

1. [xyz] matches x or y or z
2. [^pqr] matches any character _except_ p, q, or r (negation)
3. [a-zA-Z] matches a through z or A through Z, inclusive (range)

2. Groups

To operate on all items in a group, use a group expression. You can use a group expression to apply an operator to a group or to find a specified text before or after each member of the group, for example. The grouping operator is the parentheses, while the "|" is used to divide the elements.

For example:

- **My (red|pink|blue) dress**
  - Here "My red dress", "My blue kite", and "My pink dress" match the expression. "My yellow kite" or "My dress" do not match.

Let's look into some of these operators.

Group	Description
[ ]	Characters in brackets are matched.
[^]	Characters not in brackets are matched.
|	Either, or
( )	Capturing the group

More examples:

1. x(yz) parentheses create a capturing group with value yz
2. p(?:qr)* using ?: you disable the capturing group
3. a(?<ok>bc) using ?<ok> we put a name to the group

3. Quantifiers

Quantifiers specify how many characters or expressions should be matched.

Quantifiers	Description
*	0 or more (Kleene star)
+	1 or more (Kleene plus)
?	0 or 1
{ }	Exact number of characters
{min,max}	Range of characters

The quantifiers (* + {}) are also known as greedy operators. They expand the match as far as they can via the input text.

For example:

1. <[^<>]+> matches any of the characters except < or > included one or more times inside < and >
2. \w+?\d\d\w+ matches abcdef42ghijklmnfhaeij

More examples for quantifiers

x{3}	Exactly 3 of x
x{3,}	3 or more of x
x{3,6}	Between 3 and 6 of x
x*	Greedy quantifier
x*?	Lazy quantifier
x*+	Possessive quantifier

4. Anchors

Based on the current position in the string, determines whether the match will succeed or fail.

Anchors	Description
^	Beginning of the string
$	End of the string
\A	The match is at the beginning of the string.
\G	Beginning of the match
\Z	At the end of the string or before \n at the end of the string, the match occurs.
\z	Absolute end of the string
\B	No word boundary
\b	Word boundary

For example:

1. ^\d{3} matched 444 in 444-888-999-..
2. -\d{3}\Z matches -220 in 110-220
3. \babc\b performs a "whole words only" search
4. \Bend\w*\b matches "ends", "ender" in "end sends endure lender"
5. ^Hello me$ matches the string Hello me

Now, that you have got some brief idea about the characters, quantifiers and groups. Let’s look into some combined examples.

1. ^(\d*)[.,](\d+)$ matches numbers like 12,3 or 12.3
2. ^[a-zA-Z0-9]*$ matches any alphanumeric with spaces.
3. ^[\s]*(.*?)[\s]*$ matches the text by avoiding the extra spaces
4. ^([a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6})*$ could be used for matching email
5. (https?)://(www)?.?(\\w+).(\\w+)/?(\\w+)? can be used matching URLs

Advanced Regex Commands

5. Lookarounds

When the regex engine processes the lookaround expression, it first creates a substring from the present place to the beginning (lookbehind) or end (lookahead) of the original string, and then runs Regex. With the use of the lookaround pattern, IsMatch on that picked substring. A positive or negative assertion might be used to assess the outcome's success.

Lookaround	Description
(?=check)	Positive Lookahead
(?!check)	Negative Lookahead
(?<=check)	Positive Lookbehind
(?<!check)	Negative Lookbehind

Examples:

1. (?=\d{10})\d{4} matches 2348 in 2348856787
2. (?<=\d)rat matches mat in 2mat
3. (?!theatre)the\w+ matches theme
4. \w{3}(?<!mon)ster matches munster

6. POSIX commands

A character class is a combination of a small number of characters and a large number of characters. Only within bracket expressions can we use POSIX character classes. To generate regular expressions, the POSIX standard supports the following character classes.

POSIX	Description
[:alpha:]	PCRE (C, PHP, R…): ASCII letters A-Z and a-z
[:alpha:]	Ruby 2: Unicode letter or ideogram
[:alnum:]	PCRE (C, PHP, R…): ASCII digits and letters A-Z and a-z
[:alnum:]	Ruby 2: Unicode digit, letter or ideogram
[:punct:]	PCRE (C, PHP, R…): ASCII punctuation mark
[:punct:]	Ruby: Unicode punctuation mark

Examples:

1. [8[:alpha:]]+ sample match could beWellDone88
2. [[:alpha:]\d]+ sample match could bекошка99
3. [[:alnum:]]{10} sample match could beABC1275851
4. [[:alnum:]]{10} sample match could be кошка67810
5. [[:punct:]]+ sample match could be ?!.,:;
6. [[:punct:]]+ sample match could be ?,:˜\}

7. Unicode property escapes

The following are examples of Unicode property escapes:

1. \p{prop=value}: All characters with the prop property have the value value.
2. \P{prop=value}: All characters without a property prop with the value value are matched. Match \p{bin_prop}: all characters with the bin prop binary property set to True.
3. \P{bin_prop}: All characters with the binary attribute bin prop set to False will be matched.

8. Flags

A flag is a parameter that can be added to a regex to change how it searches. A flag modifies a regular expression's default searching behaviour. It does a regex search in a unique method. A single lowercase alphabetic character is used to represent a flag. There are six flags in the JavaScript regex, each providing a different purpose.

Flag	Description
i	Makes the expression search case-insensitively.
g	Makes the expression search for all occurrences.
s	Makes a wild character. match newlines as well.
m	Instead of matching the beginning and conclusion of the entire string, the boundary characters ^ and $ match the beginning and ending of each individual line.
y	Starts the expression's search from the index specified by the lastIndex attribute.
u	Assumes that individual characters are code points rather than code units, and so matches 32-bit characters.

When employing the forward slashes / to build an expression, flags come after the second slash. This can be expressed in general notation as follows: `\pattern\flag`

For example, if the flag i was added to the regex /a/, the result would be /a/i.

To provide a regex with many flags, we write them one by one (without any spaces or other delimiters).

If we gave the flags i and g to the regex /a/, for example, we'd write /a/ig (or equivalently /a/gi, as the order doesn't matter).

> Note: The sequence in which flags appear is irrelevant; flags simply change the behaviour of searching, thus placing one before the other makes no difference.

9. Recurse

The following engines enable recursion: PCRE (C, PHP, R...) Perl Ruby 2+ Python via the alternate regex package JGSoft (not available in a programming language)

The most common application of recursion is to match balanced constructions.

Command	Description
(?R)	Recurse entire pattern
(?1)	Recurse the first subpattern
(?+1)	Recurse first relative subpattern
(?&name)	Recurse subpattern name
(?P=name)	Match subpattern name
(?P>name)	Recurse subpattern name

a(?R)?z, a(?0)?z, and a\g<0>?z are all regexes that match one or more letters followed by the same number of letters z.

Examples:

- (\((?R)?\)) match parentheses like ((()))
- (\((?R)*\)) match parentheses like (()()())
- \w{3}\d{4}(?R)? matches patterns like ccc8888ggg9999

Possible Performance pitfalls

10. Regular Expression Performance Pitfalls

Because two "equivalent" regexes might have substantial changes in processing performance, you should understand how your regex engine works.

1. It is feasible to construct regexes that match in exponential time, but you must essentially TRY to do so.
2. Regexes that run in quadratic time are more commonly created by accident.
3. Problems of many kinds
    - Recompilation (from forgetting to compile regexes used multiple times)
    - The Middle Dot-star (which causes backtracking)
        - The first approach, use a character class that is negated.
        - Use reluctant quantifiers as a second option.

Tips to increase the performance

11. Improving the performance of regular expressions

- When you require parentheses but not capture, use non-capturing groups.
- Do a quick check before attempting a match, if the regex is very complex, e.g.
    - Does an email address contain '@'?
- Present the most likely option(s) first, e.g.
    - light green|dark green|brown|yellow|green|pink leaf
- Minimize the amount of looping
    - \d\d\d\d\d\d is faster than \d{6}
    - aaaaaa+ is faster than a{6,}
- Avoid obvious backtracking, e.g.
    - Mr|Ms|Mrs should be M(?:rs?|s)
    - Good night|Good morning should be Good (?:night|morning)

Conclusion

In this blog, you came across some of the interesting regex command concepts that would be very helpful in the matching of different kinds of strings. You also learned about some of the pitfalls that could happen when using regex and that could affect the performance of your application too. Further to get over these pitfalls, the blog also discusses some of the common tips that could help you in overcoming these obstacles.

Useful Resources

- Technical Interview Questions
- Coding Interview Questions
- Interview Resources
- DSA- Programming
- Mock Interview

Links to More Interview Questions

C Interview Questions	Php Interview Questions	C Sharp Interview Questions
Web Api Interview Questions	Hibernate Interview Questions	Node Js Interview Questions
Cpp Interview Questions	Oops Interview Questions	Devops Interview Questions
Machine Learning Interview Questions	Docker Interview Questions	Mysql Interview Questions
Css Interview Questions	Laravel Interview Questions	Asp Net Interview Questions
Django Interview Questions	Dot Net Interview Questions	Kubernetes Interview Questions
Operating System Interview Questions	React Native Interview Questions	Aws Interview Questions
Git Interview Questions	Java 8 Interview Questions	Mongodb Interview Questions
Dbms Interview Questions	Spring Boot Interview Questions	Power Bi Interview Questions
Pl Sql Interview Questions	Tableau Interview Questions	Linux Interview Questions
Ansible Interview Questions	Java Interview Questions	Jenkins Interview Questions