RegEX cheatsheet

A quick reference for regular expressions (regex), including symbols, ranges, grouping, assertions and some sample patterns to get you started.

Getting Started

Introduction

This is a quick cheat sheet to getting started with regular expressions.

- Regex in Python (quickref.me)
- Regex in JavaScript (quickref.me)
- Regex in PHP (quickref.me)
- Regex in Java (quickref.me)

Character Classes

`[abc]`	A single character of: a, b or c
`[^abc]`	A character except: a, b or c
`[a-z]`	A character in the range: a-z
`[^a-z]`	A character not in the range: a-z

Quantifiers

`a?`	Zero or one of a
`a*`	Zero or more of a
`a+`	One or more of a
`[0-9]+`	One or more of 0-9
`a{3}`	Exactly 3 of a
`a{3,}`	3 or more of a

- Regex in MySQL (quickref.me)
- Regex in Vim (quickref.me)
- Regex in Emacs (quickref.me)
- Online regex tester (regex101.com)

Common Metacharacters

^	{	+
<	[	*
)	>	.
(	|	$
\	?	

Escape these special characters with \

Substitution

\0	Complete match contents
\1	Contents in capture group 1

[0-9]	A digit in the range: 0-9
[a-zA-Z]	A character in the range: a-z or A-Z
[a-zA-Z0-9]	A character in the range: a-z, A-Z or 0-9

Meta Sequences

.	Any single character
\s	Any whitespace character
\S	Any non-whitespace character
\d	Any digit, Same as [0-9]
\D	Any non-digit, Same as [^0-9]
\w	Any word character
\W	Any non-word character
\X	Any Unicode sequences, linebreaks included
\C	Match one data unit
\R	Unicode newlines

a{3,6}	Between 3 and 6 of a
a*	Greedy quantifier
a*?	Lazy quantifier
a*+	Possessive quantifier

Anchors

\G	Start of match
^	Start of string
$	End of string
\A	Start of string
\Z	End of string
\z	Absolute end of string
\b	A word boundary
\B	Non-word boundary

Group Constructs

(...)	Capture everything enclosed
(a|b)	Match either a or b

`$1`	Contents in capture group 1
`${foo}`	Contents in capture group `foo`
`\x20`	Hexadecimal replacement values
`\x{06fa}`	Hexadecimal replacement values
`\t`	Tab
`\r`	Carriage return
`\n`	Newline
`\f`	Form-feed
`\U`	Uppercase Transformation
`\L`	Lowercase Transformation
`\E`	Terminate any Transformation

Assertions

`(?(1)yes|no)`	Conditional statement
`(?(R)yes|no)`	Conditional statement

`\v`	Vertical whitespace character
`\V`	Negation of \v - anything except newlines and vertical tabs
`\h`	Horizontal whitespace character
`\H`	Negation of \h
`\K`	Reset match
`\n`	Match nth subpattern
`\pX`	Unicode property X
`\p{...}`	Unicode property or script category
`\PX`	Negation of \pX
`\P{...}`	Negation of \p
`\Q...\E`	Quote; treat as literals
`\k<name>`	Match subpattern name
`\k'name'`	Match subpattern name
`\k{name}`	Match subpattern name
`\gn`	Match nth subpattern

`(?:...)`	Match everything enclosed
`(?>...)`	Atomic group (non-capturing)
`(?|...)`	Duplicate subpattern group number
`(?#...)`	Comment
`(?'name'...)`	Named Capturing Group
`(?<name>...)`	Named Capturing Group
`(?P<name>...)`	Named Capturing Group
`(?imsxXU)`	Inline modifiers
`(?(DEFINE)...)`	Pre-define patterns before using them

Lookarounds

`(?=...)`	Positive Lookahead
`(?!...)`	Negative Lookahead
`(?<=...)`	Positive Lookbehind

`(?(R#)yes|no)`	Recursive Conditional statement
`(?(R&name)yes|no)`	Conditional statement
`(?(?=...)yes|no)`	Lookahead conditional
`(?(?<=...)yes|no)`	Lookbehind conditional

Flags/Modifiers	
`g`	Global
`m`	Multiline
`i`	Case insensitive
`x`	Ignore whitespace
`s`	Single line
`u`	Unicode
`X`	eXtended
`U`	Ungreedy
`A`	Anchor

`\g{n}`	Match nth subpattern
`\g<n>`	Recurse nth capture group
`\g'n'`	Recurses nth capture group.
`\g{-n}`	Match nth relative previous subpattern
`\g<+n>`	Recurse nth relative upcoming subpattern
`\g'+n'`	Match nth relative upcoming subpattern
`\g'letter'`	Recurse named capture group `letter`
`\g{letter}`	Match previously-named capture group `letter`
`\g<letter>`	Recurses named capture group `letter`
`\xYY`	Hex character YY
`\x{YYYY}`	Hex character YYYY
`\ddd`	Octal character ddd
`\cY`	Control character Y

`(?<!...)`	Negative Lookbehind

Lookaround lets you match a group before (lookbehind) or after (lookahead) your main pattern without including it in the result.

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

`J`	Duplicate group names

`[\b]`	Backspace character

POSIX Character Classes

Character Class	Same as	Meaning
`[[:alnum:]]`	`[0-9A-Za-z]`	Letters and digits
`[[:alpha:]]`	`[A-Za-z]`	Letters
`[[:ascii:]]`	`[\x00-\x7F]`	ASCII codes 0-127
`[[:blank:]]`	`[\t ]`	Space or tab only
`[[:cntrl:]]`	`[\x00-\x1F\x7F]`	Control characters
`[[:digit:]]`	`[0-9]`	Decimal digits
`[[:graph:]]`	`[[:alnum:][:punct:]]`	Visible characters (not space)
`[[:lower:]]`	`[a-z]`	Lowercase letters
`[[:print:]]`	`[ -~] == [ [:graph:]]`	Visible characters
`[[:punct:]]`	`[!"#$%&'()*+,-./:;<=>?@[]^_`{|}~]`	Visible punctuation characters
`[[:space:]]`	`[\t\n\v\f\r ]`	Whitespace
`[[:upper:]]`	`[A-Z]`	Uppercase letters
`[[:word:]]`	`[0-9A-Za-z_]`	Word characters

Control verb

`(*ACCEPT)`	Control verb
`(*FAIL)`	Control verb
`(*MARK:NAME)`	Control verb
`(*COMMIT)`	Control verb
`(*PRUNE)`	Control verb
`(*SKIP)`	Control verb
`(*THEN)`	Control verb
`(*UTF)`	Pattern modifier
`(*UTF8)`	Pattern modifier
`(*UTF16)`	Pattern modifier
`(*UTF32)`	Pattern modifier
`(*UCP)`	Pattern modifier

Character Class	Same as	Meaning
[[:xdigit:]]	[0-9A-Fa-f]	Hexadecimal digits
[[:<:]]	[\b(?=\w)]	Start of word
[[:>:]]	[\b(?<=\w)]	End of word

(*CR)	Line break modifier
(*LF)	Line break modifier
(*CRLF)	Line break modifier
(*ANYCRLF)	Line break modifier
(*ANY)	Line break modifier
\R	Line break modifier
(*BSR_ANYCRLF)	Line break modifier
(*BSR_UNICODE)	Line break modifier
(*LIMIT_MATCH=x)	Regex engine modifier
(*LIMIT_RECURSION=d)	Regex engine modifier
(*NO_AUTO_POSSESS)	Regex engine

modifier

Regex

Regex examples

Characters

`ring`	Match ring springboard etc.
`.`	Match a, 9, + etc.
`h.o`	Match hoo, h2o, h/o etc.
`ring\?`	Match ring?
`\(quiet\)`	Match (quiet)
`c:\\windows`	Match c:\windows

Use \ to search for these special characters:
`[ \ ^ $ . | ? * + ( ) { }`

Alternatives

`cat|dog`	Match cat or dog
`id|identity`	Match id or identity
`identity|id`	Match id or identity

Order longer to shorter when alternatives overlap

Character classes

`[aeiou]`	Match any vowel
`[^aeiou]`	Match a NON vowel
`r[iau]ng`	Match ring, wrangle, sprung, etc.
`gr[ae]y`	Match gray or grey
`[a-zA-Z0-9]`	Match any letter or digit
`[\u3a00-\ufa99]`	Match any Unicode Hàn ( )

In `[ ]` always escape `. \ ]` and sometimes `^ - .`

Shorthand classes

`\w`	"Word" character (letter, digit, or

Occurrences

`colou?r`	Match color or colour

Greedy versus lazy

`* + {n,}` greedy	Match as much as possible

	underscore)
`\d`	Digit
`\s`	Whitespace (space, tab, vtab, newline)
`\W, \D, or \S`	Not word, digit, or whitespace
`[\D\S]`	Means not digit or whitespace, both match
`[^\d\s]`	Disallow digit and whitespace

`[BW]ill[ieamy's]*`	Match Bill, Willy, William's etc.
`[a-zA-Z]+`	Match 1 or more letters
`\d{3}-\d{2}-\d{4}`	Match a SSN
`[a-z]\w{1,7}`	Match a UW NetID

`<.+>`	Finds 1 big match in bold
`*? +? {n,}?` lazy	Match as little as possible
`<.+?>`	Finds 2 matches in bold

Scope

`\b`	"Word" edge (next to non "word" character)
`\bring`	Word starts with "ring", ex ringtone
`ring\b`	Word ends with "ring", ex spring
`\b9\b`	Match single digit 9, not 19, 91, 99, etc..
`\b[a-zA-Z]{6}\b`	Match 6-letter words
`\B`	Not word edge
`\Bring\B`	Match springs and wringer

Modifiers

`(?i)`[a-z]*`(?-i)`	Ignore case ON / OFF
`(?s).*(?-s)`	Match multiple lines (causes . to match newline)
`(?m)^.*$(?-m)`	^ & $ match lines not whole string
`(?x)`	#free-spacing mode, this EOL comment ignored

`^\d*$`	Entire string must be digits
`^[a-zA-Z]{4,20}$`	String must have 4-20 letters
`^[A-Z]`	String must begin with capital letter

`(?-x)`	free-spacing mode OFF
/regex/`ismx`	Modify mode for entire string

Groups

`(in|out)put`	Match input or output
`\d{5}(-\d{4})?`	US zip code ("+ 4" optional)

Parser tries EACH alternative if match fails after group.
Can lead to catastrophic backtracking.

Back references

`(to) (be) or not \1 \2`	Match to be or not to be
`([^\s])\1{2}`	Match non-space, then same twice more aaa, ...
`\b(\w+)\s+\1\b`	Match doubled words

Non-capturing group

`on(?:click|load)`	Faster than `on(click|load)`

Use non-capturing or atomic groups when possible

Atomic groups

`(?>red|green|blue)`	Faster than non-capturing
`(?>id|identity)\b`	Match id, but not

Lookaround

`(?= )`	Lookahead, if you can find ahead
`(?! )`	Lookahead,if you can not find ahead
`(?<= )`	Lookbehind, if you can find behind

identity

"id" matches, but \b fails after atomic group, parser doesn't backtrack into group to retry 'identity'

If alternatives overlap, order longer to shorter.

If-then-else

Match "Mr." or "Ms." if word "her" is later in string

```
M(?(?=.*?\bher\b)s|r)\.
```

requires lookaround for IF condition

(?<!)	Lookbehind, if you can NOT find behind
\b\w+?(?=ing\b)	Match warbling, string, fishing, ...
\b(?!\w+ing\b)\w+\b	Words NOT ending in "ing"
(?<=\bpre).*?\b	Match pretend, present, prefix, ...
\b\w{3}(?<!pre)\w*?\b	Words NOT starting with "pre"
\b\w+(?<!ing)\b	Match words NOT ending in "ing"

RegEx in Python

Getting started

Import the regular expressions module

```
import re
```

Functions

Examples

re.search()

```
>>> sentence = 'This is a sample string'
>>> bool(re.search(r'this', sentence, flags=re.I))
True
>>> bool(re.search(r'xyz', sentence))
False
```

re.findall	Returns a list containing all matches
re.finditer	Return an iterable of match objects (one for each match)
re.search	Returns a Match object if there is a match anywhere in the string
re.split	Returns a list where the string has been split at each match
re.sub	Replaces one or many matches with a string
re.compile	Compile a regular expression pattern for later use
re.escape	Return string with all non-alphanumerics

Flags

re.I	re.IGNORECASE	Ignore case
re.M	re.MULTILINE	Multiline
re.L	re.LOCALE	Make \w,\b,\s

re.findall()

```
>>> re.findall(r'\bs?pare?\b', 'par spar apparent spare part pare')
['par', 'spar', 'spare', 'pare']
>>> re.findall(r'\b0*[1-9]\d{2,}\b', '0501 035 154 12 26 98234')
['0501', '154', '98234']
```

re.finditer()

```
>>> m_iter = re.finditer(r'[0-9]+', '45 349 651 593 4 204')
>>> [m[0] for m in m_iter if int(m[0]) < 350]
['45', '349', '4', '204']
```

re.split()

```
>>> re.split(r'\d+', 'Sample123string42with777numbers')
['Sample', 'string', 'with', 'numbers']
```

re.sub()

```
>>> ip_lines = "catapults\nconcatenate\ncat"
>>> print(re.sub(r'^', r'* ', ip_lines, flags=re.M))
* catapults
* concatenate
* cat
```

re.compile()

```
>>> pet = re.compile(r'dog')
>>> type(pet)
<class '_sre.SRE_Pattern'>
>>> bool(pet.search('They bought a dog'))
True
```

		locale dependent
re.S	re.DOTALL	Dot matches all (including newline)
re.U	re.UNICODE	Make \w,\b,\d,\s unicode dependent
re.X	re.VERBOSE	Readable style

```
>>> bool(pet.search('A cat crossed their path'))
False
```

Regex in JavaScript

test()

```
let textA = 'I like APPles very m
let textB = 'I like APPles';
let regex = /apples$/i

// Output: false
console.log(regex.test(textA));

// Output: true
console.log(regex.test(textB));
```

search()

```
let text = 'I like APPles very mu
let regexA = /apples/;
let regexB = /apples/i;

// Output: -1
console.log(text.search(regexA));

// Output: 7
console.log(text.search(regexB));
```

exec()

```
let text = 'Do you like apples?';
let regex= /apples/;

// Output: apples
console.log(regex.exec(text)[0]);

// Output: Do you like apples?
console.log(regex.exec(text).inpu
```

match()

split()

```
let text = 'Here are apples and a
let regex = /apples/gi;

// Output: [ "apples", "apPleS" ]
console.log(text.match(regex));
```

```
let text = 'This 593 string will be brok294en at places where d1gits are
let regex = /\d+/g

// Output: [ "This ", " string will be brok", "en at places where d", "g
console.log(text.split(regex))
```

matchAll()

```
let regex = /t(e)(st(\d?))/g;
let text = 'test1test2';
let array = [...text.matchAll(reg

// Output: ["test1", "e", "st1",
console.log(array[0]);

// Output: ["test2", "e", "st2",
console.log(array[1]);
```

replace()

```
let text = 'Do you like
aPPles?';
let regex = /apples/i

// Output: Do you like mangoes?
let result = text.replace(regex,
'mangoes');
console.log(result);
```

replaceAll()

```
let regex = /apples/gi;
let text = 'Here are apples and a

// Output: Here are mangoes and m
let result = text.replaceAll(rege
console.log(result);
```

Regex in PHP

Functions

preg_match()	Performs a regex match
preg_match_all()	Perform a global regular expression match
preg_replace_callback()	Perform a regular expression search and replace using a callback

preg_replace

```
$str = "Visit Microsoft!";
$regex = "/microsoft/i";

// Output: Visit QuickRef!
echo preg_replace($regex,
"QuickRef", $str);
```

`preg_replace()`	Perform a regular expression search and replace
`preg_split()`	Splits a string by regex pattern
`preg_grep()`	Returns array entries that match a pattern

preg_match

```php
$str = "Visit QuickRef";
$regex = "#quickref#i";

// Output: 1
echo preg_match($regex, $str);
```

preg_grep

```php
$arr = ["Jane", "jane", "Joan", "
$regex = "/Jane/";

// Output: Jane
echo preg_grep($regex, $arr);
```

preg_matchall

```php
$regex = "/[a-zA-Z]+ (\d+)/";
$input_str = "June 24, August 13, and December 30";
if (preg_match_all($regex, $input_str, $matches_out)) {

    // Output: 2
    echo count($matches_out);

    // Output: 3
    echo count($matches_out[0]);

    // Output: Array("June 24", "August 13", "December 30")
    print_r($matches_out[0]);

    // Output: Array("24", "13", "30")
    print_r($matches_out[1]);
}
```

preg_split

```php
$str = "Jane\tKate\nLucy Marion";
$regex = "@\s@";
```

```
// Output: Array("Jane", "Kate", "Lucy", "Marion")
```

Regex in Java

Styles

First way

```
Pattern p = Pattern.compile(".s", Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("aS");
boolean s1 = m.matches();
System.out.println(s1);   // Outputs: true
```

Second way

```
boolean s2 = Pattern.compile("[0-9]+").matcher("123").matches();
System.out.println(s2);   // Outputs: true
```

Third way

```
boolean s3 = Pattern.matches(".s", "XXXX");
System.out.println(s3);   // Outputs: false
```

Pattern Fields

CANON_EQ	Canonical equivalence
CASE_INSENSITIVE	Case-insensitive matching
COMMENTS	Permits whitespace and comments
DOTALL	Dotall mode
MULTILINE	Multiline mode
UNICODE_CASE	Unicode-aware case folding
UNIX_LINES	Unix lines mode

Methods

Pattern

- Pattern compile(String regex [, int flags])

Examples

Replace sentence:

```
String regex = "[A-Z\n]{5}$";
String str = "I like APP\nLE";
```

- boolean matches([String regex,] CharSequence input)
- String[] split(String regex [, int limit])
- String quote(String s)

Matcher

- int start([int group | String name])
- int end([int group | String name])
- boolean find([int start])
- String group([int group | String name])
- Matcher reset()

String

- boolean matches(String regex)
- String replaceAll(String regex, String replacement)
- String[] split(String regex[, int limit])

```
Pattern p = Pattern.compile(regex, Pattern.MULTILINE);
Matcher m = p.matcher(str);

// Outputs: I like Apple!
System.out.println(m.replaceAll("pple!"));
```

Array of all matches:

```
String str = "She sells seashells by the Seashore";
String regex = "\\w*se\\w*";

Pattern p = Pattern.compile(regex, Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher(str);

List<String> matches = new ArrayList<>();
while (m.find()) {
    matches.add(m.group());
}

// Outputs: [sells, seashells, Seashore]
System.out.println(matches);
```

Regex in MySQL

Functions

REGEXP	Whether string matches regex
REGEXP_INSTR()	Starting index of substring matching regex (NOTE: Only MySQL 8.0+)
REGEXP_LIKE()	Whether string matches regex (NOTE: Only MySQL 8.0+)
REGEXP_REPLACE()	Replace substrings matching regex (NOTE: Only MySQL 8.0+)
REGEXP_SUBSTR()	Return substring matching regex (NOTE: Only MySQL 8.0+)

REGEXP

```
expr REGEXP pat
```

Examples

```
mysql> SELECT 'abc' REGEXP '^[a-d]';
1
mysql> SELECT name FROM cities WHERE name REGEXP '^A
mysql> SELECT name FROM cities WHERE name NOT REGEXP
mysql> SELECT name FROM cities WHERE name REGEXP 'A|
mysql> SELECT 'a' REGEXP 'A', 'a' REGEXP BINARY 'A';
1   0
```

REGEXP_REPLACE

```
REGEXP_REPLACE(expr, pat, repl[, pos[, occurrence[,
match_type]]])
```

Examples

```
mysql> SELECT REGEXP_REPLACE('a b c', 'b', 'X');
a X c
mysql> SELECT REGEXP_REPLACE('abc ghi', '[a-z]+', 'X
abc X
```

REGEXP_SUBSTR

```
REGEXP_SUBSTR(expr, pat[, pos[, occurrence[,
match_type]]])
```

Examples

```
mysql> SELECT REGEXP_SUBSTR('abc def ghi', '[a-z]+');
abc
mysql> SELECT REGEXP_SUBSTR('abc def ghi', '[a-z]+',
ghi
```

REGEXP_LIKE

```
REGEXP_LIKE(expr, pat[, match_type])
```

REGEXP_INSTR

```
REGEXP_INSTR(expr, pat[, pos[, occurrence[,
```

Examples

```
mysql> SELECT regexp_like('aba', 'b+')
1
mysql> SELECT regexp_like('aba', 'b{2}')
0
mysql> # i: case-insensitive
mysql> SELECT regexp_like('Abba', 'ABBA', 'i');
1
mysql> # m: multi-line
mysql> SELECT regexp_like('a\nb\nc', '^b$', 'm');
1
```

Examples

```
mysql> SELECT regexp_instr('aa aaa aaaa', 'a{3}');
2
mysql> SELECT regexp_instr('abba', 'b{2}', 2);
2
mysql> SELECT regexp_instr('abbabba', 'b{2}', 1, 2);
5
mysql> SELECT regexp_instr('abbabba', 'b{2}', 1, 3,
7
```

Related Cheatsheet

Grep Cheatshee
Quick Reference

Recent Cheatsheet

Remote Work Re
Quick Reference

Homebrew Chea
Quick Reference

PyTorch Cheatsl
Quick Reference

Taskset Cheatsh
Quick Reference

QuickRef.ME

Share quick reference and cheat sheet for developers.

#Notes