

RegEX Manual

version 1.0

This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).



Table of contents

1. Introduction.....	3
2. General.....	4
2.1. Regular Expression's Engines.....	4
2.2. Notation conventions and modes.....	5
2.3. Useful resources.....	5
3. Characters.....	6
3.1. Literal characters.....	6
3.2. Metacharacters.....	6
3.3. Wild character.....	6
3.4. Escaping metacharacters.....	7
3.5. Other special characters.....	7
4. Character sets.....	8
4.1. Character ranges.....	8
4.2. Negative character sets.....	9
4.3. Metacharacters inside character sets.....	9
4.4. Shorthand characters set.....	10
5. Repetition.....	11
5.1. Repetition Metacharacters.....	11
5.2. Quantified repetition.....	12
5.3. Greedy expressions.....	12
5.4. Lazy expressions.....	13
6. Grouping and alternation.....	14
6.1. Grouping.....	14
6.2. Alternation metacharacter.....	15
6.2.1. Eager, greedy or lazy.....	15
6.2.2. Efficiency when using alternation.....	16
7. Anchors (and word boundaries).....	17
7.1. Start and End anchors.....	17
7.2. Line breaks and multiline.....	18
7.3. Word boundaries.....	19

8. Look behind & look forward.....	20
8.1. Look behind.....	20
8.2. Look forward.....	20
8.3. Look behind & forward.....	20
Annex A – RegEx in Notepad++.....	21

1. Introduction

Regular Expressions aka. [regex](#)* or [regexp](#) are not a programming language but are used by a lot of programming languages, text editors, etc.

Typical usages:

- Test if a phone number has the correct number of digits
- Test if an e-mail address is in a valid format
- Search a document for either “behavior” or “behaviour”
- Replace all occurrences of “Bob” or “Bobby” with “Robert”
- Count how many times “lessons” is preceded by “computer”, “video” or “online”

*: in this manual, I have chosen to use following syntax for [regex](#): [RegEx](#)

2. General

2.1. Regular Expression's Engines

Following programming and database languages (not exhaustive list) allow to perform searches based on regular expressions:

C/C++, Java, JavaScript, .NET, Perl, PHP, Python, Ruby, Apache, MySQL

Most text editors also have a [RegEx](#) implementation.

[RegEx](#) may be implemented with slight differences between all those languages.

There are online [RegEx](#) websites where [RegEx](#) expressions may be used. They are perfect for training or test purposes. Most known [RegEx](#) websites are:

www.regexr.com

www.regex101.com

www.regexpal.com

Each of these website allows to select the “flavor” of [RegEx](#) implementation, I.e. : JavaScript, PCRE*, Python, Golang, grep**, etc.

*PCRE: **P**erl **C**ompatible **R**egular **E**xpressions, a library written in C, which implements a regular expression engine, inspired by the capabilities of the Perl programming language.

grep: **Global **R**egular **E**xpression **P**rint, in Unix/Linux worlds, it is a command-line utility for searching plain-text data sets for lines that match a regular expression.

2.2. Notation conventions and modes

Not all [RegEx](#) implementations require start and end forward slashes “/” as delimiters of the regular expression (e.g. `/abc/`). See [Annex A](#), for the implementation in Notepad++

In this manual, we use the syntax with start and end forward slashes, which are used by most online [RegEx](#) websites.

Modes:

Standard: `/regex/` Stops once the first occurrence is found
Global: `/regex/g` Find all occurrences throughout the document.
Case insensitive: `/regex/i`
Multiline: `/regex/m`



[RegEx](#) searches are always performed from left to right



Best practice is not to enable case insensitive mode.
There are other better techniques.

2.3. Useful resources

This manual is mostly based on the personal notes I took while following the excellent LinkedIn online tutorial “[Learning Regular Expressions](#)” by Kevin Skoglund.

The best online overview of RegEx I have ever come across is certainly the Youtube vid: “[Learn Regular Expressions in 20 Minutes](#)” by [Web Dev Simplified Channel](#)

3. Characters

3.1. Literal characters

The most basic search that can be performed. Similar to the “find” search that can be made on any software (text editor, MS Office, etc.)

The typed string is searched in the document. e.g.: `/text/`

3.2. Metacharacters

Metacharacters are characters with a special meaning:

- they transform literal characters into powerful expressions.
- they can have more than one more meaning, depending on the syntax/context.
- there are only a few to learn:

`\ . * + - { } [] ^ $ | ? () : < ! =`

3.3. Wild character

Certainly the most important metacharacter.

`/./` any single character except line returns. (point and space are considered as characters)

Example:

`/4.00/` matches “4.00” but also “4700”, “4 00” and “4-00”



A good regular expression should match the text you want to target and only that text, nothing more.

3.4. Escaping metacharacters

To use a metacharacter in the search as literal character, it must be escaped.

It is done by using the backslash character “\” just before the metacharacter to be escaped.

- only metacharacters should be escaped.
- literal characters should never be escaped, it may give them meaning (e.g. `/t/`)
- backslash “\” itself can be escaped by a backslash: `\\`
- quotations marks are not metacharacters. They do not need to be escaped.

Example:

`/4\\.00/` matches “4.00” but neither “4700” nor “4-00”

3.5. Other special characters

Space: `/ /`

Tabulation: `/t/`

Line returns:

Carriage return: `/r/`

Line feed: `/n/`

Carriage return + Line feed: `/r\n/`



Line returns depend on the OS the file has been created on:

MS Windows: `\r\n`

Unix: `\n`

Mac OS: `\r`

4. Character sets

Character set matches anyone of several characters but only one character.
The order of characters in the set doesn't matter.

Examples:

`/[aeiou]/` matches any vowel (only one character)

`/[br]ed/` matches strings: “**bed**” and “**red**”

4.1. Character ranges

A character range is a character set that includes all characters between two defined characters, separated by dash “-” character.

Examples:

`/[0-9]/` matches any single digit between 0 and 9 included

`/[A-Za-z]/` matches any single letter, capital or small

Warning:

`/[50-99]/` is not all numbers between 50 and 99. It is equivalent to `/[0-9]/`
(practically, it represents: 5, 0-9, 9)



Don't forget: character sets always target a single character.

“-” is only a metacharacter inside a character set. Otherwise it is a literal dash character.

4.2. Negative character sets

Circumflex character “^” inside a character set (and always at the beginning) negates the character set. It matches none of the characters **but will still look for a character !**

Examples:

`/see[^mn]/` matches “seek”, “sees”, “see.” and “see ” (point and space are still considered as characters)

But it doesn't match “seem”, “seen” nor “see” (remember, it still looks for a character !)

If circumflex character “^” is not placed at the beginning inside the character set, it is automatically escaped.

Example:

`/t[ia^o]c/` matches “tic”, “tac”, “t^c” and “toc” !

4.3. Metacharacters inside character sets

Most metacharacters inside a character set are automatically escaped. Therefore, they do not need to be escaped again.

Examples:

`/h[a.]t/` matches “hat” and “h.t”, but not “hot”

There are exceptions: `]-\^` (the latter one if at the beginning, otherwise escaped. See here above)

Examples:

`/var[[\][0-9][\]]/` matches “var(2)”, “var[3]” and even “var[7]” and “var(5)”, etc.

`/file[0\-__]1/` matches “file01”, “file-1”, “file\1” and “file_1”

`/2022[-/]02[-/]10/` matches “2022-02-10”, “2022/02/10”, “2022-02/10” and “2022/02-10”

↑ ↑
In this case, it is not necessary to escape the dash because there is no character before it in the character set.

4.4. Shorthand characters set

Positive shorthand characters set:

<code>\d/</code>	digit	is equivalent to:	<code>/[0-9]/</code>
<code>\w/</code>	word character*	is equivalent to:	<code>/[A-Za-z0-9_]/</code>
<code>\s/</code>	white space	is equivalent to:	<code>/[\t\r\n]/</code>

Negative shorthand characters set:

<code>\D/</code>	not digit	is equivalent to:	<code>/[^\d-9]/</code>
<code>\W/</code>	not word character*	is equivalent to:	<code>/[^\A-Za-z0-9_]/</code>
<code>\S/</code>	not white space	is equivalent to:	<code>/[^\t\r\n]/</code>

*: doesn't match letters with accents



Underscore “_” is a word character.

Hyphen or dash “-” is not a word character.

Equivalent notations:

$[\wedge d]$	is equivalent to:	$\wedge D$
$[\wedge w]$	is equivalent to:	$\wedge W$
$[\wedge s]$	is equivalent to:	$\wedge S$

Warning:

`/[^\d\s]/` is not the same as: `/[\D\S]/`

Neither digit nor
space character

Either not digit or
not space character

5. Repetition

5.1. Repetition Metacharacters

All repetition metacharacters act on preceding item.

*	zero or more times
+	one or more times
?	zero or one time

Examples:

<code>./+ /</code>	matches any string of characters except a line return
<code>/Good .+ /</code>	matches "Good morning.", "Good day.", "Good evening." and "Good night."
<code>/\d+ /</code>	matches "10725"
<code>/\s[a-z]+ed\s /</code>	matches lowercase words, ending in "ed" (unless the word is at the beginning of first line or at the end of last line !)



For this last example, [RegEx](http://www.regexr.com) implementation of www.regexr.com also ignores words at the beginning of each line...

<code>/apples* /</code>	matches "apple", "apples" and "appless"
<code>/apples+ /</code>	matches "apples", "appless" but not "apple"
<code>/apples? /</code>	matches "apple" and "apples" but not "appless"
<code>/behaviour?r /</code>	matches "behaviour" and "behavior"

5.2. Quantified repetition

Use of curly braces:

{ starts quantified repetition of preceding item
} ends quantified repetition of preceding item

Syntax: {min,max}

min and max are positive numbers

min must always be included and can be zero

max is optional

This leads to three different syntaxes:

`\d{4,8}/` matches numbers with four to eight digits
`\d{4}/` matches numbers with exactly four digits (min is max)
`\d{4,}/` matches numbers with four or more digits (infinite)

Examples:

`/0\d{2} \d{3} \d{2} \d{2}/` matches most swiss phone numbers (e.g. 032 499 88 00)
`\d{3}-\d{3}-\d{4}/` matches most US phone numbers (e.g. 555-013-0420)
`/A{1,2} bonds/` matches "A bonds" and "AA bonds" but not "AAA bonds"

5.3. Greedy expressions

Standard repetition quantifiers are greedy (that means that [RegEx](#) tries to match the longest possible string):

`/"\.", "\.", "\."/` on string text ""Milton", "Waddams", "Initech, Inc"" may apply to:

"Milton", "Waddams"
or "Milton", "Waddams", "Initech, Inc"
or "Milton", "Waddams", "Initech, Inc"

But, by default, RegEx will match the whole string (one of last two possibilities)

5.4. Lazy expressions

The use of “?” character makes precedent quantifier lazy, i.e. it instructs quantifier to use a “lazy strategy” for making choices.

“?” has already been seen before (see [#5.1.Repetition Metacharacters](#)), but here, it is in another context.

Possible syntaxes are:

`/*?/`

`/+?/`

`{min,max}?/`

`/??/`

For the latter syntax (`/??/`):

the first interrogation mark means: “0 or 1 occurrence”

the second interrogation mark means: “lazy strategy”

Lazy expressions:

- still defer to overall match
- are not necessarily faster or slower

Using the example from previous chapter:

`/.+?/, ".+?"/` on string text ““Milton”, “Waddams”, “Initech, Inc”” matches only:

`“Milton”`, `“Waddams”`

6. Grouping and alternation

6.1. Grouping

Use of parenthesis:

- (starts grouped expression
-) ends grouped expression

Grouping:

- groups portions of the expressions
- applies repetition operators to a group
- creates a group of alternation expressions
- captures the group for use in matching and replacing

Examples:

- `/(abc)+/` matches "abc", "abcabc", and "abcabcabc", and etc.
- `/(in)?dependent/` matches "independent" and "dependent"
- `/run(s)?/` is the same as `/runs?/`

Example of capturing for replacing on string "555-666-7890": (on www.regexr.com)

`/(\d{3})-(\d{3})-(\d{4})/`
\$1 \$2 \$3



Depending on the online [RegEx](http://www.regexr.com) or text editor, call of a group may use another character than "\$"

Most systems allow then to replace the expression by another formatted one.
(on www.regexr.com: "replace" button)

Example:

\$1.\$2.\$3 gives back string: "555.666.7890"

6.2. Alternation metacharacter

Use of vertical bar:

| match previous or next expression. It is an OR operator.

Alternation:

either matches expression on the left or match expression on the right
ordered, leftmost expression gets precedence
multiple choices can be daisy-chained
group alternation expressions to keep them distinct

Examples:

/apple orange/	matches "apple" and "orange"
/abc def ghi jkl/	matches "abc", "def", "ghi" and "jkl"
/w(ei ie)rd/	matches "weird" and "wierd"
/(AA BB CC DD){4}/	matches "AABBAACC", "CCCCBBBB" and ...
/apple (juice sauce)/	matches "apple juice" and "apple sauce" but is not the same as:
/apple juice sauce/	matches "apple juice" and "sauce"

6.2.1. Eager, greedy or lazy

Depending on the way the [RegEx](#) is written, it will turn it eager (wants to give you quickly a result) or greedy (wants to give you the biggest result possible).

Example with the string: "peanutbutter"

eager:

/peanut|peanutbutter/ matches "peanut"

greedy:

/peanut(butter)?/ matches "peanutbutter"

lazy: (by using the last [RegEx](#))

/peanut(butter)??/ matches "peanut"

Use of lazy character "??"

6.2.2. Efficiency when using alternation

(In terms of computation time)


Always put simplest (most efficient) expression first

Example:

```
^w+_d{2,4}|\d{4}_export|export\_d{2}/
```

would be more efficiently written than:

```
/export\_d{2}|\d{4}_export|^w+_d{2,4}/
```


would give less results than

7. Anchors (and word boundaries)

7.1. Start and End anchors

Start and end anchors:

reference a position, not an actual character
are zero-width

Syntax:

<code>^</code>	start of string/line	} supported by all RegEx engines
<code>\$</code>	end of string/line	
<code>\A</code>	start of string, never start of line	} Newer: <u>not</u> supported by all RegEx engines. for instance, not support within Javascript
<code>\Z</code>	end of string, never end of line	

Example: look for string “elephant” as first word of the string

`/^elephant/` or `/\Aelephant/`



Circumflex character is used here in another context than a character set and has therefore another “effect”.

Example: look for “apple” as last word of the string

`/elephant$/` or `/elephant\Z/`

If both start and end anchors are used, it implies that expression must completely define the string we are analyzing:

`/^elephant$/` or `/\Aelephant\Z/`

An application must check e-mail addresses. In this example, let's consider the following e-mail address: john.doe@nowhere.com

A possible [RegEx](#) to identify it would be:

`\w+@\w+\.[a-z]{3}/`

But this [RegEx](#) would partially validate the following string: `john.doe@nowhere.com-junk`

To only match real e-mail addresses, let's use start and end anchors:

`/^\w+@\w+\.[a-z]{3}$/`

↑ ↑

start anchor end anchor

7.2. Line breaks and multiline

Behaviour of anchors depend on the line mode chosen (single-line mode vs. multiline mode)

Single-line mode: (by-default mode)

`^` and `$` **do not** match at line breaks
`\A` and `\Z` **do not** match at line breaks

Multiline mode:

`^` and `$` **will** match at line breaks
`\A` and `\Z` **do not** match at line breaks

Example: in single-line mode, the following [RegEx](#) `/^[a-z]+/` (with global mode on) only matches the first line (without line return) of:

```
tomatoes
bread
potatoes
strawberries
pineapple juice
```

To match every (starting) word of each line, turn multiline mode ON:

```
tomatoes
bread
potatoes
strawberries
pineapple juice
```

Multiline mode was not present at the beginning of [RegEx](#), it has been added afterwards. Today, most programming languages implements the [RegEx](#) multiline mode.

e.g.: PERL, Ruby, PHP, JavaScript, Java, Net, Python, etc.

7.3. Word boundaries

Word boundaries:

- reference a position, not an actual character
- must be placed before the first word character in the string, and:
- must be placed after the last word character in the string
- define a separation between a word character and a non-word character



Reminder: a word character belongs to set [A-Za-z0-9_]

Syntax:

- `\b` word boundary (start/end of the word)
- `\B` not a word boundary

Examples:

`^b\b+w+\b/` finds four matches in the string “You are a maid”, as well as the entire string “data_897” but finds only part of “english-speaking”.
(As a matter fact, underscore “_” is a word character, dash “-” isn’t.)

`^B\b+w+\B/` finds three matches in the string “You are a maid”.

`^b\b+w+s\b/` looks for words ending in “s”

Warning:

`^bNew\bYork\b/` doesn’t match “New York”

`^bNew\b\bYork\b/` matches “New York”

8. Look behind & look forward

Allows to find strings preceded and/or followed by a specific string.
As well as strings not preceded and/or not followed by a specific string.

Syntax:

`...(?...)` use of parenthesis, interrogation mark and ...

8.1. Look behind

Syntax:

`(?<=RegEx1)RegEx2` matches any `RegEx2` preceded by `RegEx1` (positive search)

`(?<!=RegEx1)RegEx2` matches any `RegEx2` not preceded by `RegEx1` (negative search)

Examples:

`/(?<=[tT]he).+/?` matches any string preceded by “the” or “The”

`/(?<![tT]he).+/?` matches any string not preceded by “the” or “The”

8.2. Look forward

Syntax:

`RegEx2(?=RegEx1)` matches any `RegEx2` followed by `RegEx1` (positive search)

`RegEx2(?!=RegEx1)` matches any `RegEx2` not followed by `RegEx1` (negative search)

Examples:

`/go (?=at)/` matches any “go” followed by “at”

`/go (?!at)/` matches any “go” not followed by “at”

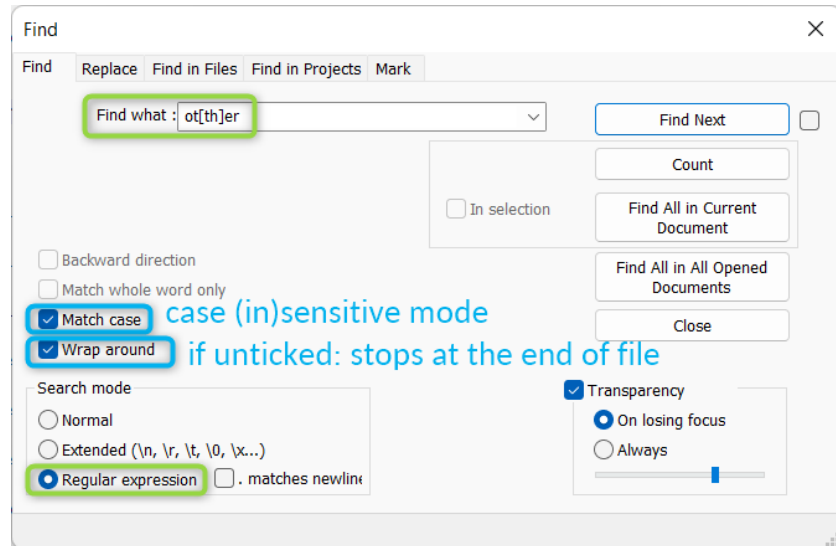
8.3. Look behind & forward

Mixed example with string “I am an astronaut (just kidding...)”

`/(?<=\\().+(?=\\))/` matches what is between parenthesis: `just kidding...`

Annex A – RegEx in Notepad++

CTRL-F or Search Menu --> Find:

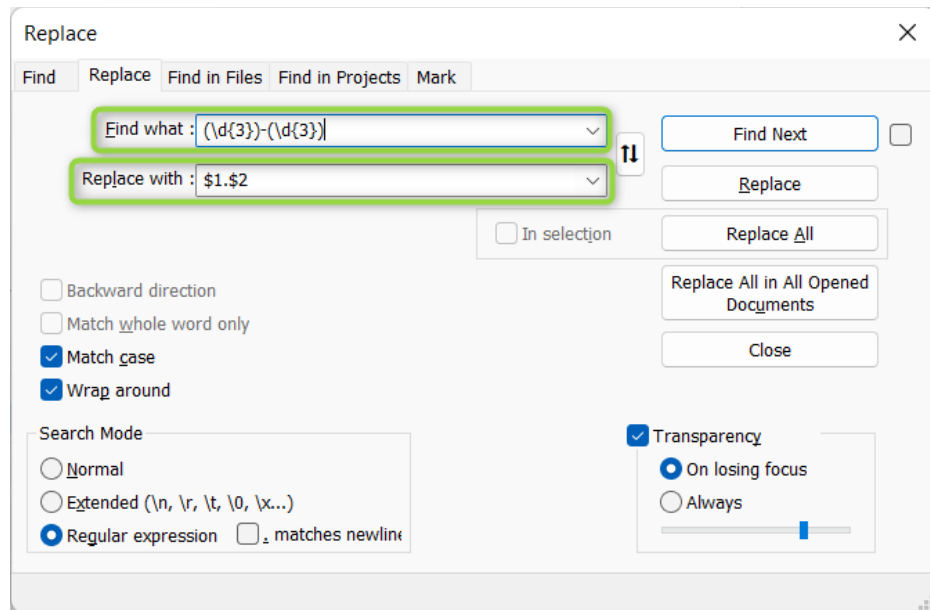


RegEx capabilities (Notepad++ v8.3.2)

Capability	Active ?	Remarks
Wild character “.”	Yes	Escaping wild character also OK
Special chars (space, tab, line returns)	Yes	--
Simple character set	Yes	--
Character range	Yes	--
Negative character set	Yes	--
Shorthand character set	Yes	--
Metacharacter inside char. set	partly	e.g. “]” doesn’t need to be escaped
Simple repetition	Yes	--
Quantified repetition	Yes	--
Greedy vs. lazy expressions	Yes	--
Grouping	Yes	Call groups for replacement with “\$” character
Alternation metacharacter	Yes	--
Start and End anchors	partly	^ \$: OK \A \Z: NOK
Line breaks and multiline	n/a	Notepad++ search is always in multiline mode
Word boundaries	Yes	--
Look behind & look forward	Yes	--

For replacement of groups:

CTRL-H or Search Menu --> Replace:



Use of character "\$" to retrieve/call found groups.