FREE eBook

LEARNING Regular Expressions

Free unaffiliated eBook created from
Stack Overflow contributors.

#regex

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: regular-expressions

It is an unofficial and free Regular Expressions ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Regular Expressions.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Regular Expressions

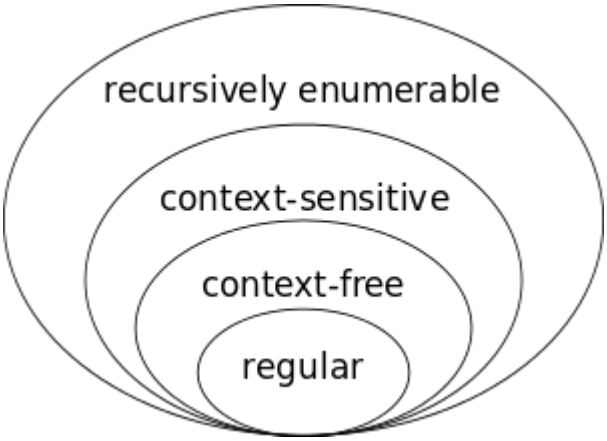
Remarks

For many programmers the *regex* is some sort of magical sword that they throw to solve any kind of text parsing situation. But this tool is nothing magical, and even though it's great at what it does, it's not a full featured programming language (*i.e.* it is **not** Turing-complete).

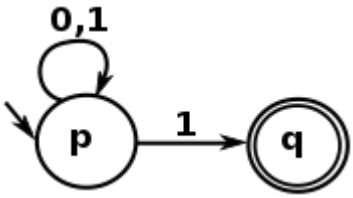
What does 'regular expression' mean?

Regular expressions express a language defined by a *regular grammar* that can be solved by a *nondeterministic finite automaton* (NFA), where matching is represented by the states.

A *regular grammar* is the most simple grammar as expressed by the *Chomsky Hierarchy*.



Simply said, a regular language is visually expressed by what an NFA can express, and here's a very simple example of NFA:



And the *Regular Expression* language is a textual representation of such an automaton. That last example is expressed by the following regex:

```
^[01]*1$
```

Which is matching any string beginning with `0` or `1`, repeating 0 or more times, that ends with a `1`. In other words, it's a regex to match odd numbers from their binary representation.

Are all regex actually a *regular* grammar?

Actually they are not. Many regex engines have improved and are using *push-down automata*, that can stack up, and pop down information as it is running. Those automata define what's called *context-free* grammars in Chomsky's Hierarchy. The most typical use of those in non-regular *regex*, is the use of a recursive pattern for parenthesis matching.

A recursive regex like the following (that matches parenthesis) is an example of such an implementation:

```
{((?>[^\(\)]+|(?R))*)}
```

(this example does not work with python's `re` engine, but with the `regex` engine, or with the PCRE engine).

Resources

For more information on the theory behind Regular Expressions, you can refer to the following courses made available by MIT:

- Automata, Computability, and Complexity
- Regular Expressions & Grammars
- Specifying Languages with Regular Expressions and Context-Free Grammars

When you're writing or debugging a complex regex, there are online tools that can help visualize regexes as automatons, like the debuggex site.

Versions

PCRE

Version	Released
2	2015-01-05
1	1997-06-01

Used by: PHP 4.2.0 (and higher), Delphi XE (and higher), Julia, Notepad++

Perl

Version	Released
1	1987-12-18
2	1988-06-05
3	1989-10-18
4	1991-03-21
5	1994-10-17
6	2009-07-28

.NET

Version	Released
1	2002-02-13
4	2010-04-12

Languages: C#

Java

Version	Released
4	2002-02-06
5	2004-10-04
7	2011-07-07
SE8	2014-03-18

JavaScript

Version	Released
1.2	1997-06-11
1.8.5	2010-07-27

Python

Version	Released
1.4	1996-10-25
2.0	2000-10-16
3.0	2008-12-03
3.5.2	2016-06-07

Oniguruma

Version	Released
Initial	2002-02-25
5.9.6	2014-12-12
Onigmo	2015-01-20

Boost

Version	Released
0	1999-12-14
1.61.0	2016-05-13

POSIX

Version	Released
BRE	1997-01-01
ERE	2008-01-01

Languages: Bash

Examples

Character Guide

Note that some syntax elements have different behavior depending on the expression.

Syntax	Description
`?`	Match the preceding character or subexpression 0 or 1 times. Also used for non-capturing groups, and named capturing groups.
`*`	Match the preceding character or subexpression 0 or more times.
`+`	Match the preceding character or subexpression 1 or more times.
`{n}`	Match the preceding character or subexpression exactly *n* times.
`{min,}`	Match the preceding character or subexpression *min* or more times.
`{,max}`	Match the preceding character or subexpression *max* or fewer times.
`{min,max}`	Match the preceding character or subexpression at least *min* times but no more than `max` times.
`-`	When included between square brackets indicates `to`; e.g. [3-6] matches characters 3, 4, 5, or 6.
`^`	Start of string (or start of line if the multiline `/m` option is specified), or negates a list of options (i.e. if within square brackets `[]`)
`$`	End of string (or end of a line if the multiline `/m` option is specified).
`(...)`	Groups subexpressions, captures matching content in special variables (`\1`, `\2`, etc.) that can be used later within the same regex, for example `(\w+)\s\1\s` matches word repetition
`(?<name>...)`	Groups subexpressions, and captures them in a named group
`(?:...)`	Groups subexpressions without capturing
`.`	Matches any character except line breaks (`\n`, and usually `\r`).
`[...]`	Any character between these brackets should be matched once. NB: `^` following the open bracket negates this effect. `-` occurring inside the brackets allows a range of values to be specified (unless it's the first or last character, in which case it just represents a regular dash).
`\`	Escapes the following character. Also used in meta sequences - regex tokens with special meaning.
`\$`	dollar (i.e. an escaped special character)
`\(`	open parenthesis (i.e. an escaped special character)
`\)`	close parenthesis (i.e. an escaped special character)

Syntax	Description
`\*`	asterisk (i.e. an escaped special character)
`\.`	dot (i.e. an escaped special character)
`\?`	question mark (i.e. an escaped special character)
`\[`	left (open) square bracket (i.e. an escaped special character)
`\\`	backslash (i.e. an escaped special character)
`\]`	right (close) square bracket (i.e. an escaped special character)
`\^`	caret (i.e. an escaped special character)
`\{`	left (open) curly bracket / brace (i.e. an escaped special character)
`|`	pipe (i.e. an escaped special character)
`\}`	right (close) curly bracket / brace (i.e. an escaped special character)
`\+`	plus (i.e. an escaped special character)
`\A`	start of a string
`\Z`	end of a string
`\z`	absolute of a string
`\b`	word (alphanumeric sequence) boundary
`\1`,`\2`, etc.	back-references to previously matched subexpressions, grouped by `()`, `\1` means the first match, `\2` means second match etc.
`[\b]`	backspace - when `\b` is inside a character class (`[]`)matches backspace
`\B`	negated `\b` - matches at any position between two-word characters as well as at any position between two non-word characters
`\D`	non-digit
`\d`	digit
`\e`	escape
`\f`	form feed
`\n`	line feed
`\r`	carriage return

Syntax	Description
`\S`	non-white-space
`\s`	white-space
`\t`	tab
`\v`	vertical tab
`\W`	non-word
`\w`	word (i.e. alphanumeric character)
`{...}`	named character set
`|`	or; i.e. delineates the prior and preceding options.

Read Getting started with Regular Expressions online:
https://riptutorial.com/regex/topic/259/getting-started-with-regular-expressions

Chapter 2: Anchor Characters: Caret (^)

Remarks

Terminology

The Caret (^) character is also referred to by the following terms:

- hat
- control
- uparrow
- chevron
- circumflex accent

Usage

It has two uses in regular expressions:

- To denote the start of the line
- If used immediately after a square bracket (`[^`) it acts to negate the set of allowed characters (i.e. `[123]` means the character 1, 2, or 3 is allowed, whilst the statement `[^123]` means any character other than 1, 2, or 3 is allowed.

Character Escaping

To express a caret without special meaning, it should be escaped by preceding it with a backslash; i.e. `\^`.

Examples

Start of Line

When multi-line `(?m)` modifier is turned off, `^` matches only the input string's beginning:

For the regex

```
^He
```

The following input strings match:

- `Hedgehog\nFirst line\nLast line`
- `Help me, please`
- `He`

And the following input strings do **not** match:

- `First line\nHedgehog\nLast line`
- `IHedgehog`
- `Hedgehog` (due to white-spaces)

When multi-line `(?m)` modifier is turned on, `^` matches every line's beginning:

```
^He
```

The above would match any input string that contains a line beginning with `He`.

Considering `\n` as the new line character, the following lines match:

- `Hello`
- `First line\nHedgehog\nLast line` (second line only)
- `My\nText\nIs\nHere` (last line only)

And the following input strings do **not** match:

- `Camden Hells Brewery`
- `Helmet` (due to white-spaces)

Matching empty lines using `^`

Another typical use case for caret is matching empty lines (or an empty string if the multi-line modifier is turned off).

In order to match an empty line (multi-line **on**), a caret is used next to a `$` which is another anchor character representing the position at the end of line (Anchor Characters: Dollar ($)). Therefore, the following regular expression will match an empty line:

```
 ^$
```

Escaping the caret character

If you need to use the `^` character in a character class (Character classes), either put it somewhere other than the beginning of the class:

```
[12^3]
```

Or escape the `^` using a backslash `\`:

```
[\^123]
```

If you want to match the caret character itself outside a character class, you need to escape it:

```
\^
```

This prevents the `^` being interpreted as the anchor character representing the beginning of the string/line.

Comparison start of line anchor and start of string anchor

While many people think that `^` means the start of a string, it actually means start of a line. For an actual start of string anchor use, `\A`.

The string `hello\nworld` (or more clearly)

```
hello
world
```

Would be matched by the regular expressions `^h`, `^w` and `\Ah` but not by `\Aw`

Multiline modifier

By default, the caret `^` metacharacter matches the **position** before the first character in the string.

Given the string "**charsequence**" applied against the following patterns: `/^char/` & `/^sequence/`, the engine will try to match as follows:

- `/^char/`
    - **^** - charsequence
    - **c** - `c`harsequence
    - **h** - `ch`arsequence
    - **a** - `cha`rsequence
    - **r** - `char`sequence

  Match Found

- `/^sequence/`
    - **^** - charsequence
    - **s** - charsequence

  Match not Found

The same behaviour will be applied even if the string contains *line terminators*, such as `\r?\n`. Only the position at the start of the string will be matched.

For example:

```
/^/g
```

┊ char\r\n
\r\n
sequence

However, if you need to match after every line terminator, you will have to set the **multiline** mode (`//m`, `(?m)`) within your pattern. By doing so, the caret `^` will match "the beginning of each line", which corresponds to the position at the beginning of the string and the positions **immediately after**[1] the line terminators.

[1] In some flavors (Java, PCRE, ...), `^` will not match after the line terminator, if the line terminator is the last in the string.

For example:

```
/^/gm
```

┊ char\r\n
┊ \r\n
┊ sequence

Some of the regular expression engines that support Multiline modifier:

- Java

```
Pattern pattern = Pattern.compile("(?m)^abc");
Pattern pattern = Pattern.compile("^abc", Pattern.MULTILINE);
```

- .NET

```
var abcRegex = new Regex("(?m)^abc");
var abdRegex = new Regex("^abc", RegexOptions.Multiline)
```

- PCRE

```
/(?m)^abc/
/^abc/m
```

- Python 2 & 3 (built-in `re` module)

```
abc_regex = re.compile("(?m)^abc");
abc_regex = re.compile("^abc", re.MULTILINE);
```

Read Anchor Characters: Caret (^) online: https://riptutorial.com/regex/topic/452/anchor-characters--caret----

Chapter 3: Anchor Characters: Dollar ($)

Remarks

A great deal of regex engines use a "multi-line" mode in order to search several lines in a file independently.

Therefore when using `$`, these engines will match all lines' endings. However, engines that do not use this kind of multi-line mode will only match the last position of the string provided for the search.

Examples

Match a letter at the end of a line or string

```
g$
```

The above matches one letter (the letter `g`) at the end of a *string* in most regex engines (not in Oniguruma, where the `$` anchor matches the end of a line by default, and the `m` (*MULTILINE*) modifier is used to make a `.` match any characters including line break characters, as a DOTALL modifier in most other NFA regex flavors). The `$` anchor will match the first occurrence of a `g` letter before the end of the following strings:

In the following sentences, only the letters in **bold** match:

> Anchors are characters that, in fact, do not match any character in a strin**g**
>
> Their goal is to match a specific position in that string.
>
> Bob was helpin**g**
>
> But his edit introduced examples that were not matching!

In most regular expression flavors, the `$` anchor can also match before a newline character or line break character (sequence), in a *MULTILINE* mode, where `$` matches at the end of every line instead of only at the end of a string. For example, using `g$` as our regex again, in multiline mode, the italicised characters in the following string would match:

```
tvxlt obofh necpu riist g\n aelxk zlhdx lyogu vcbke pzyay wtsea wbrju jztg\n drosf ywhed bykie lqmzg wgyhc lg\n qewrx ozrvm jwenx
```

Read Anchor Characters: Dollar ($) online: https://riptutorial.com/regex/topic/1603/anchor-characters--dollar----

Chapter 4: Atomic Grouping

Introduction

Regular non-capturing groups allow the engine to re-enter the group and attempt to match something different (such as a different alternation, or match fewer characters when a quantifier is used).

Atomic groups differ from regular non-capturing groups in that backtracking is forbidden. Once the group exits, all backtracking information is discarded, so no alternate matches can be attempted.

Remarks

A possessive quantifier behaves like an atomic group in that the engine will be unable to backtrack over a token or group.

The following are equivalent in terms of functionality, although some will be faster than others:

```
a*+abc
(?>a*)abc
(?:a+)*+abc
(?:a)*+abc
(?:a*)*+abc
(?:a*)++abc
```

Examples

Grouping with (?>)

Using an Atomic Group

Atomic groups have the format `(?>...)` with a `?>` after the open paren.

Consider the following sample text:

```
ABC
```

The regex will attempt to match starting at position 0 of the text, which is before the `A` in `ABC`.

If a case-insensitive expression `(?>a*)abc` were used, the `(?>a*)` would match 1 `A` character, leaving

```
BC
```

as the remaining text to match. The `(?>a*)` group is exited, and `abc` is attempted on the remaining

text, which fails to match.

The engine is unable to backtrack into the atomic group, and so the current pass fails. The engine moves to the next position in the text, which would be at position 1, which is after the `A` and before the `B` of `ABC`.

The regex `(?>a*)abc` is attempted again, and `(?>a*)` matches `A` 0 times, leaving

```
BC
```

as the remaining text to match. The `(?>a*)` group is exited and `abc` is attempted, which fails.

Again, the engine is unable to backtrack into the atomic group, and so the current pass fails. The regex will continue to fail until all positions in the text have been exhausted.

Using a Non-Atomic Group

Regular non-capturing groups have the format `(?:...)` with a `?:` after the open paren.

Given the same sample text, but with the case-insensitive expression `(?:a*)abc` instead, a match would occur since backtracking is allowed to occur.

At first, `(?:a*)` will consume the letter `A` in the text

```
ABC
```

leaving

```
BC
```

as the remaining text to match. The `(?:a*)` group is exited, and `abc` is attempted on the remaining text, which fails to match.

The engine backtracks into the `(?:a*)` group and attempts to match 1 fewer character: Instead of matching 1 `A` character, it attempts to match 0 `A` characters, and the `(?:a*)` group is exited. This leaves

```
ABC
```

as the remaining text to match. The regex `abc` is now able to successfully match the remaining text.

Other Example Text

Consider this sample text, with both atomic and non-atomic groups (again, case-insensitive):

```
AAAABC
```

The regex will attempt to match starting at position 0 of the text, which is before the first `A` in `AAAABC`.

The pattern using the atomic group `(?>a*)abc` will be **unable** to match, behaving almost identically to the atomic `ABC` example above: all 4 of the `A` characters are first matched with `(?>a*)` (leaving `BC` as the remaining text to match), and `abc` is unable to match on that text. The group **is not** able to be re-entered, so the match fails.

The pattern using the non-atomic group `(?:a*)abc` will be **able** to match, behaving similarly to the non-atomic `ABC` example above: all 4 of the `A` characters are first matched with `(?:a*)` (leaving `BC` as the remaining text to match), and `abc` is unable to match on that text. The group **is** able to be re-entered, so one fewer `A` is attempted: 3 `A` characters are matched instead of 4 (leaving `ABC` as the remaining text to match), and `abc` is able to successfully match on that text.

Read Atomic Grouping online: https://riptutorial.com/regex/topic/8770/atomic-grouping

Chapter 5: Back reference

Examples

Basics

Back references are used to match the same text previously matched by a capturing group. This both helps in reusing previous parts of your pattern and in ensuring two pieces of a string match.

For example, if you are trying to verify that a string has a digit from zero to nine, a separator, such as hyphens, slashes, or even spaces, a lowercase letter, another separator, then another digit from zero to nine, you could use a regex like this:

```
[0-9][-/ ][a-z][-/ ][0-9]
```

This would match `1-a-4`, but it would *also* match `1-a/4` or `1 a-4`. If we want the separators to match, we can use a capture group and a back reference. The back reference will look at the match found in the indicated capture group, and ensure that the location of the back reference matches exactly.

Using our same example, the regex would become:

```
[0-9]([-/ ])[a-z]\1[0-9]
```

The `\1` denotes the first capture group in the pattern. With this small change, the regex now matches `1-a-4` or `1 a 4` but not `1 a-4` or `1-a/4`.

The number to use for your back reference depends on the location of your capture group. The number can be from one to nine and can be found by counting your capture groups.

```
([0-9])([-/ ])[a-z][-/ ]([0-9])
|--1--||--2--|           |--3--|
```

Nested capture groups change this count slightly. You first count the exterior capture group, then the next level, and continue until you leave the nest:

```
(([0-9])([-/ ]))([a-z])
 |--2--||--3--|
|-------1------||--4--|
```

Ambiguous Backreferences

Problem: You need to match text of a certain format, for example:

```
1-a-0
6/p/0
4 g 0
```

That's a digit, a separator (one of `-`, `/`, or a space), a letter, the same separator, and a zero.

Naïve solution: Adapting the regex from the Basics example, you come up with this regex:

```
[0-9]([-/ ])[a-z]\10
```

But that probably won't work. Most regex flavors support more than nine capturing groups, and very few of them are smart enough to realize that, since there's only one capturing group, `\10` must be a backreference to group 1 followed by a literal `0`. Most flavors will treat it as a backreference to group 10. A few of those will throw an exception because there is no group 10; the rest will simply fail to match.

There are several ways to avoid this problem. One is to use named groups (and named backreferences):

```
[0-9](?<sep>[-/ ])[a-z]\k<sep>0
```

If your regex language supports it, the format `\g{n}` (where `n` is a number) can enclose the backreference number in curly brackets to separate it from any digits after it:

```
[0-9]([-/ ])[a-z]\g{1}0
```

Another way is to use extended regex formatting, separating the elements with insignificant whitespace (in Java you'll need to escape the space in the brackets):

```
(?x) [0-9] ([-/ ]) [a-z] \1 0
```

If your regex flavor doesn't support those features, you can add unnecessary but harmless syntax, like a non-capturing group:

```
[0-9]([-/ ])[a-z](?:\1)0
```

...or a dummy quantifier (this is possibly the only circumstance in which `{1}` is useful):

```
[0-9]([-/ ])[a-z]\1{1}0
```

Read Back reference online: https://riptutorial.com/regex/topic/4072/back-reference

Chapter 6: Backtracking

Examples

What causes Backtracking?

To find a match, the regex engine will consume characters one by one. When a partial match begins, the engine will remember the start position so it can go back in case the following characters don't complete the match.

- If the match is complete, the is no backtracking
- If the match isn't complete, the engine will backtrack the string (like when you rewind an old tape) to try to find a whole match.

For example: `\d{3}[a-z]{2}` against the string `abc123def` will be browsed as such:

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does match \d (third one)
abc123def
      ^ Does match [a-z] (first one)
abc123def
       ^ Does match [a-z] (second one)
          MATCH FOUND
```

Now lets change the regex to `\d{2}[a-z]{2}` against the same string (`abc123def`):

```
abc123def
^ Does not match \d
abc123def
 ^ Does not match \d
abc123def
  ^ Does not match \d
abc123def
   ^ Does match \d (first one)
abc123def
    ^ Does match \d (second one)
abc123def
     ^ Does not match [a-z]
abc123def
    ^ BACKTRACK to catch \d{2} => (23)
abc123def
      ^ Does match [a-z] (first one)
abc123def
```

```
      ^ Does match [a-z] (second one)
          MATCH FOUND
```

Why can backtracking be a trap?

Backtracking can be caused by optional quantifiers or alternation constructs, because the regex engine will try to explore every path. If you run the regex `a+b` against `aaaaaaaaaaaaa` there is no match and the engine will find it pretty fast.

But if you change the regex to `(aa*)+b` the number of combinations will grow pretty fast, and most (not optimized) engines will try to explore all the paths and will take an eternity to try to find a match or throw a timeout exception. This is called **catastrophic backtracking**.

Of course, `(aa*)+b` seems a newbie error but it's here to illustrate the point and sometimes you'll end up with the same issue but with more complicated patterns.

A more extreme case of catastrophic backtracking occurs with the regex `(x+x+)+y` (you've probably seen it before here and here), which needs exponential time to figure out that a string that contains `x`s and nothing else (e.g `xxxxxxxxxxxxxxxxxxx`) don't match it.

How to avoid it?

Be as specific as possible, reduce as much as possible the possible paths. Note that some regex matchers are not vulnerable to backtracking, such as those included in `awk` or `grep` because they are based on Thompson NFA.

Read Backtracking online: https://riptutorial.com/regex/topic/977/backtracking

Chapter 7: Capture Groups

Examples

Basic Capture Groups

A *group* is a section of a regular expression enclosed in parentheses `()`. This is commonly called "sub-expression" and serves two purposes:

- It makes the sub-expression atomic, i.e. it will either match, fail or repeat as a whole.
- The portion of text it matched is accessible in the remainder of the expression and the rest of the program.

Groups are numbered in regex engines, starting with 1. Traditionally, the maximum group number is 9, but many modern regex flavors support higher group counts. Group 0 always matches the entire pattern, the same way surrounding the entire regex with brackets would.

The ordinal number increases with each opening parenthesis, regardless of whether the groups are placed one-after-another or nested:

```
foo(bar(baz)?) (qux)+|(bla)
   1   2       3      4
```

groups and their numbers

After an expression achieves an overall match, all of its groups will be in use - whether a particular group has managed to match anything or not.

A group can be optional, like `(baz)?` above, or in an alternative part of the expression that was not used of the match, like `(bla)` above. In these cases, non-matching groups simply won't contain any information.

If a quantifier is placed behind a group, like in `(qux)+` above, the overall group count of the expression stays the same. If a group matches more than once, its content will be the last match occurrence. However, modern regex flavors allow accessing all sub-match occurrences.

If you wished to retrieve the date and error level of a log entry like this one:

```
2012-06-06 12:12.014 ERROR: Failed to connect to remote end
```

You could use something like this:

```
^(\d{4}-\d{2}-\d{2}) \d{2}:\d{2}.\d{3} (\w*): .*$
```

This would extract the date of the log entry `2012-06-06` as capture group 1 and the error level `ERROR`

as capture group 2.

Backreferences and Non-Capturing Groups

Since Groups are "numbered" some engines also support matching what a group has previously matched again.

Assuming you wanted to match something where two equals strings of length three are divided by a `$` you'd use:

```
(.{3})\$\1
```

This would match any of the following strings:

```
"abc$abc"
"a b$a b"
"af $af "
"   $   "
```

If you want a group to not be numbered by the engine, You may declare it non-capturing. A non-capturing group looks like this:

```
(?:)
```

They are particularly useful to repeat a certain pattern any number of times, since a group can also be used as an "atom". Consider:

```
(\d{4}(?:-\d{2}){2} \d{2}:\d{2}.\d{3}) (.*)[\r\n]+\1 \2
```

This will match two logging entries in the adjacent lines that have the same timestamp and the same entry.

Named Capture Groups

Some regular expression flavors allow *named capture groups*. Instead of by a numerical index you can refer to these groups by name in subsequent code, i.e. in backreferences, in the replace pattern as well as in the following lines of the program.

Numerical indexes change as the number or arrangement of groups in an expression changes, so they are more brittle in comparison.

For example, to match a word (`\w+`) enclosed in either single or double quotes (`['"]`), we could use:

```
(?<quote>['"])\w+\k{quote}
```

Which is equivalent to:

```
(['"])\w+\1
```

In a simple situation like this a regular, numbered capturing group does not have any draw-backs.

In more complex situations the use of named groups will make the structure of the expression more apparent to the reader, which improves maintainability.

Log file parsing is an example of a more complex situation that benefits from group names. This is the Apache Common Log Format (CLF):

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

The following expression captures the parts into named groups:

```
(?<ip>\S+) (?<logname>\S+) (?<user>\S+) (?<time>\[[^]]+\]) (?<request>"[^"]+") (?<status>\S+)
(?<bytes>\S+)
```

The syntax depends on the flavor, common ones are:

- `(?<name>...)`
- `(?'name'...)`
- `(?P<name>...)`

Backreferences:

- `\k<name>`
- `\k{name}`
- `\k'name'`
- `\g{name}`
- `(?P=name)`

In the .NET flavor you can have several groups sharing the same name, they will use capture stacks.

In PCRE you have to explicitly enable it by using the `(?J)` modifier (`PCRE_DUPNAMES`), or by using the branch reset group `(?|)`. Only the last captured value will be accessible though.

```
(?J)(?<a>...)(?<a>...)
(?|(?<a>...)|(?<a>...))
```

Read Capture Groups online: https://riptutorial.com/regex/topic/660/capture-groups

Chapter 8: Character classes

Remarks

Simple classes

Regex	Matches
`[abc]`	Any of the following characters: `a`, `b`, or `c`
`[a-z]`	Any character from `a` to `z`, **inclusive** (this is called a *range*)
`[0-9]`	Any digit from `0` to `9`, **inclusive**

Common classes

Some groups/ranges of characters are so often used, they have special abbreviations:

Regex	Matches
`\w`	Alphanumeric characters plus the underscore (also referred to as "word characters")
`\W`	Non-word characters (same as `[^\w]`)
`\d`	Digits (*wider* than `[0-9]` since include Persian digits, Indian ones etc.)
`\D`	Non-digits (*shorter* than `[^0-9]` since reject Persian digits, Indian ones etc.)
`\s`	Whitespace characters (spaces, tabs, etc...) **Note**: may vary depending on your engine/context
`\S`	Non-whitespace characters

Negating classes

A caret (^) after the opening square bracket works as a negation of the characters that follow it. This will match all characters that are not in the character class.

Negated character classes also match line break characters, therefore if these are not to be matched, the specific line break characters must be added to the class (\r and/or \n).

Regex	Matches
`[^AB]`	Any character **other** than `A` and `B`
`[^\d]`	Any character, **except** digits

Examples

The basics

Suppose we have a list of teams, named like this: `Team A`, `Team B`, ..., `Team Z`. Then:

- `Team [AB]`: This will match either either `Team A` or `Team B`
- `Team [^AB]`: This will match any team **except** `Team A` or `Team B`

We often need to match characters that "belong" together in some context or another (like letters from `A` through `Z`), and this is what character classes are for.

Match different, similar words

Consider the character class `[aeiou]`. This character class can be used in a regular expression to match a set of similarly spelled words.

`b[aeiou]t` matches:

- bat
- bet
- bit
- bot
- but

It does not match:

- bout
- btt
- bt

Character classes on their own match one and only one character at a time.

Non-alphanumerics matching (negated character class)

```
[^0-9a-zA-Z]
```

This will match all characters that are neither numbers nor letters (alphanumerical characters). If the underscore character _ is also to be negated, the expression can be shortened to:

```
[^\w]
```

Or:

```
\W
```

In the following sentences:

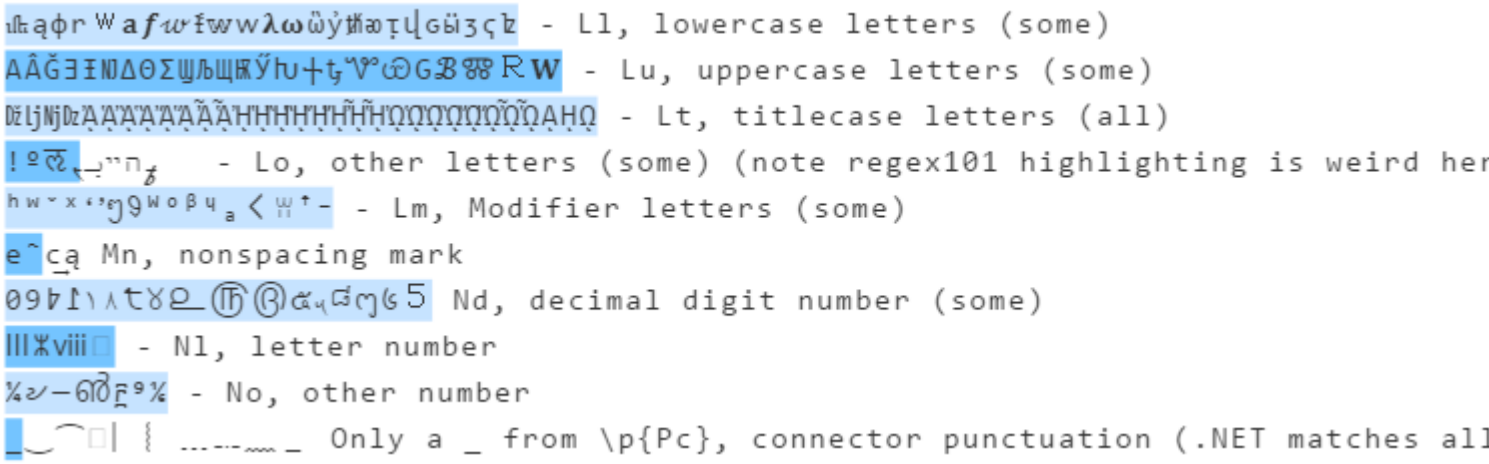
1. Hi, what's up?
2. I can't wait for 2017!!!

The following characters match:

1. `,`, `,`, `'`, `?` and the end of line character.
2. `'`, `,`, `!` and the end of line character.

UNICODE NOTE

Note that some flavors with Unicode character properties support may interpret `\w` and `\W` as `[\p{L}\p{N}_]` and `[^\p{L}\p{N}_]` which means other Unicode letters and numeric characters will be included as well (see PCRE docs). Here is a PCRE `\w` test:



In .NET, `\w` = `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Mn}\p{Nd}\p{Pc}]`, and note it does not match `\p{Nl}` and `\p{No}` unlike PCRE (see the `\w` .NET documentation):

```
^\w+
```

Input

```
\w = [\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Lm}\p{Mn}\p{Nd}\p{Pc}] (no \p{Nl},
\p{No} as PCRE)
- Ll, lowercase letters (some)
- Lu, uppercase letters (some)
- Lt, titlecase letters (all)
- Lo, other letters (some)
- Lm, Modifier letters (some)
Mn, nonspacing mark (some)
Nd, decimal digit number (some)
- Pc, connector punctuation
```

Note that for some reason, Unicode 3.1 lowercase letters (like) are not matched.

Java's `(?U)\w` will match a mix of what `\w` matches in PCRE and .NET:

```
(?mU)^\w+
```

```
- Ll, lowercase letters
- Lu, uppercase letters
- Lt, titlecase letters
- Lo, other letters (some)
- Lm, Modifier letters (some)
Mn, nonspacing mark (some)
Nd, decimal digit number
- Nl, letter number (SOME ARE MATCHED)
- No, other number (NO MATCH)
- Pc, connector punctuation
```

Non-digits matching (negated character class)

```
[^0-9]
```

This will match all characters that are not ASCII digits.

If Unicode digits are also to be negated, the following expression can be used, depending on your flavor/language settings:

```
[^\d]
```

This can be shortened to:

```
\D
```

You may need to enable Unicode character properties support explicitly by using the `u` modifier or

programmatically in some languages, but this may be non-obvious. To convey the intent explicitly, the following construct can be used (when support is available):

```
\P{N}
```

Which *by definition* means: any character which is not a numeric character in any script. In a negated character range, you may use:

```
[^\p{N}]
```

In the following sentences:

1. Hi, what's up?
2. I can't wait for 2017!!!

The following characters will be matched:

1. `,`, , `'`, `?`, the end of line character and all letters (lowercase and uppercase).
2. `'`, , `!`, the end of line character and all letters (lowercase and uppercase).

Character class and common problems faced by beginner

1. Character Class

Character class is denoted by `[]`. Content inside a character class is treated as `single character separately`. e.g. suppose we use

```
[12345]
```

In the example above, it means match `1 or 2 or 3 or 4 or 5` . In simple words, it can be understood as `or condition for single characters` (**stress on single character**)

1.1 Word of caution

- In character class, there is no concept of matching a string. So, if you are using regex `[cat]`, it does not mean that it should match the word `cat` literally but it means that it should match either `c` or `a` or `t`. This is a very common misunderstanding existing among people who are newer to regex.
- Sometimes people use `|` (alternation) inside character class thinking it will act as `OR condition` which is wrong. e.g. using `[a|b]` actually means match `a` or `|` (literally) or `b`.

2. Range in character class

Range in character class is denoted using `-` sign. Suppose we want to find any character within English alphabets `A` to `Z`. This can be done by using the following character class

```
[A-Z]
```

This could be done for any valid ASCII or unicode range. Most commonly used ranges include `[A-Z]`, `[a-z]` or `[0-9]`. Moreover these ranges can be combined in character class as

```
[A-Za-z0-9]
```

This means that match any character in the range `A to Z` or `a to z` or `0 to 9`. The ordering can be anything. So the above is equivalent to `[a-zA-Z0-9]` as long as the range you define is correct.

2.1 Word of caution

- Sometimes when writing ranges for `A` to `Z` people write it as `[A-z]`. This is wrong in most cases because we are using `z` instead of `Z`. So this denotes match any character from ASCII range `65` (of A) to `122` (of z) which includes many unintended character after ASCII range `90` (of Z). **HOWEVER**, `[A-z]` can be used to match all `[a-zA-Z]` letters in POSIX-style regex when collation is set for a particular language. `[[ "ABCEDEF[]_abcdef" =~ ([A-z]+) ]] && echo "${BASH_REMATCH[1]}"` on Cygwin with `LC_COLLATE="en_US.UTF-8"` yields `ABCEDF`. If you set `LC_COLLATE` to `C` (on Cygwin, done with `export`), it will give the expected `ABCEDEF[]_abcdef`.

- Meaning of `-` inside character class is special. It denotes range as explained above. *What if we want to match `-` character literally?* We can't put it anywhere otherwise it will denote ranges if it is put between two characters. In that case we have to put `-` in starting of character class like `[-A-Z]` or in end of character class like `[A-Z-]` or `escape it` if you want to use it in middle like `[A-Z\-a-z]`.

3. Negated character class

Negated character class is denoted by `[^..]`. The caret sign `^` denotes match any character except the one present in character class. e.g.

```
[^cat]
```

means match any character except `c` or `a` or `t`.

3.1 Word of caution

- The meaning of caret sign `^` maps to negation only if its in the starting of character class. If its anywhere else in character class it is treated as literal caret character without any special meaning.
- Some people write regex like `[^]`. In most regex engines, this gives an error. The reason being when you are using `^` in the starting position, it expects at least one character that should be negated. In *JavaScript* though, this is a valid construct matching *anything but nothing*, i.e. matches any possible symbol (but diacritics, at least in ES5).

POSIX Character classes

POSIX character classes are predefined sequences for a certain set of characters.

Character class	Description
`[:alpha:]`	Alphabetic characters
`[:alnum:]`	Alphabetic characters and digits
`[:digit:]`	Digits
`[:xdigit:]`	Hexadecimal digits
`[:blank:]`	Space and Tab
`[:cntrl:]`	Control characters
`[:graph:]`	Visible characters (anything except spaces and control characters)
`[:print:]`	Visible characters and spaces
`[:lower:]`	Lowercase letters
`[:upper:]`	Uppercase letters
`[:punct:]`	Punctuation and symbols
`[:space:]`	All whitespace characters, including line breaks

Additional character classes my be available depending on the implementation and/or locale.

Character class	Description
`[:<:]`	Beginning of word
`[:>:]`	End of word
`[:ascii:]`	ASCII Characters
`[:word:]`	Letters, digits and underscore. Equivalent to `\w`

To use the inside a bracket sequence (aka. character class), you should also include the square brackets. Example:

```
[[:alpha:]]
```

This will match one alphabetic character.

```
[[:digit:]-]{2}
```

This will match 2 characters, that are either digits or `-`. The following will match:

- `--`

- `11`
- `-2`
- `3-`

More information is available on: Regular-expressions.info

Read Character classes online: https://riptutorial.com/regex/topic/1757/character-classes

Chapter 9: Escaping

Examples

Raw String Literals

It's best for readability (and your sanity) to avoid escaping the escapes. That's where raw strings literals come in. (Note that some languages allow delimiters, which are preferred over strings usually. But that's another section.)

They usually work the same way as this answer describes:

> [A] backslash, `\`, is taken as meaning "just a backslash" (except when it comes right before a quote that would otherwise terminate the literal) -- no "escape sequences" to represent newlines, tabs, backspaces, form-feeds, and so on.

Not all languages have them, and those that do use varying syntax. C# actually calls them *verbatim string literals*, but it's the same thing.

Python

```
pattern = r"regex"
```

```
pattern = r'regex'
```

C++ (11+)

The syntax here is extremely versatile. The only rule is to use a delimiter that does not appear anywhere in the regex. If you do that, no additional escaping is necessary for anything in the string. Note that the parenthesis `()` are not part of the regex:

```
pattern = R"delimiter(regex)delimiter";
```

VB.NET

Just use a normal string. Backslashes are ALWAYS literals.

C#

```
pattern = @"regex";
```

Note that this syntax also allows `""` (two double quotes) as an escaped form of `"`.

Strings

In most programming languages, in order to have a backslash in a string generated from a string literal, each backslash must be doubled in the string literal. Otherwise, it will be interpreted as an escape for the next character.

Unfortunately, any backslash required by the regex must be a literal backslash. This is why it becomes necessary to have "escaped escapes" (`\\`) when regexes are generated from string literals.

In addition, quotes (`"` or `'`) in the string literal may need to be escaped, depending on which surround the string literal. In some languages, it is possible to use either style of quotes for a string (choose the most readable one for escaping the entire string literal).

In some languages (e.g.: Java <=7), regexes cannot be expressed directly as literals such as `/\w/`; they must be generated from strings, and normally string literals are used - in this case, `"\\w"`. In these cases, literal characters such as quotes, backslashes, etc. need to be escaped. The easiest way to accomplish this may be by using a tool (like *RegexPlanet*). This specific tool is designed for Java, but it will work for any language with a similar string syntax.

What characters need to be escaped?

Character escaping is what allows certain characters (reserved by the regex engine for manipulating searches) to be literally searched for and found in the input string. Escaping depends on context, therefore this example does not cover string or delimiter escaping.

Backslashes

Saying that backslash is the "escape" character is a bit misleading. Backslash escapes and backslash brings; it actually toggles on or off the metacharacter vs. literal status of the character in front of it.

In order to use a literal backslash anywhere in a regex, it must be escaped by another backslash.

Escaping (outside character classes)

There are several characters that need to be escaped to be taken literally (at least outside char classes):

- Brackets: `[]`
- Parentheses: `()`
- Curly braces: `{}`
- Operators: `*`, `+`, `?`, `|`

- Anchors: `^`, `$`
- Others: `.`, `\`
- In order to use a literal `^` at the start or a literal `$` at the end of a regex, the character must be escaped.
- Some flavors only use `^` and `$` as metacharacters when they are at the start or end of the regex respectively. In those flavors, no additional escaping is necessary. It's usually just best to escape them anyway.

Escaping within Character Classes

- It's best practice to escape square brackets (`[` and `]`) when they appear as literals in a char class. Under certain conditions, it's not required, depending on the flavor, but it harms readability.
- The caret, `^`, is a meta character when put as the first character in a char class: `[^aeiou]`. Anywhere else in the char class, it is just a literal character.
- The dash, `-`, is a meta character, unless it's at the beginning or end of a character class. If the first character in the char class is a caret `^`, then it will be a literal if it is the second character in the char class.

Escaping the Replacement

There are also rules for escaping within the replacement, but none of the rules above apply. The only metacharacters are `$` and `\`, at least when `$` can be used to reference capture groups (like `$1` for group 1). To use a literal `$`, escape it: `\$5.00`. Likewise `\`: `C:\\Program Files\\`.

BRE Exceptions

While ERE (extended regular expressions) mirrors the typical, Perl-style syntax, BRE (basic regular expressions) has significant differences when it comes to escaping:

- There is different shorthand syntax. All of the `\d`, `\s`, `\w` and so on is gone. Instead, it has its own syntax (which POSIX confusingly calls "character classes"), like `[:digit:]`. These constructs must be within a character class.
- There are few metacharacters (`.`, `*`, `^`, `$`) that can be used normally. ALL of the other metacharacters must be escaped differently:

Braces `{}`

- `a{1,2}` matches `a{1,2}`. To match either `a` or `aa`, use `a\{1,2\}`

Parentheses `()`

- `(ab)\1` is invalid, since there is no capture group 1. To fix it and match `abab` use `\(ab\)\1`

Backslash

- Inside char classes (which are called bracket expressions in POSIX), backslash is not a metacharacter (and does not need escaping). `[\d]` matches either `\` or `d`.
- Anywhere else, escape as usual.

Other

- `+` and `?` are literals. If the BRE engine supports them as metacharacters, they must be escaped as `\?` and `\+`.

/Delimiters/

Many languages allow regex to be enclosed or delimited between a couple of specific characters, usually the forward slash `/`.

Delimiters have an impact on escaping: if the delimiter is `/` and the regex needs to look for `/` literals, then the forward slash must be escaped before it can be a literal (`\/`).

Excessive escaping harms readability, so it's important to consider the available options:

Javascript is unique because it allows forward slash as a delimiter, but nothing else (although it does allow stringified regexes).

Perl1

Perl, for example, allows almost anything to be a delimiter. Even Arabic characters:

```
$str =~ m ش ش
```

Specific rules are mentioned in Perl's documentation.

PCRE allows two types of delimiters: matched delimiters and bracket-style delimiters. Matched delimiters make use of a single character's pair, while bracket-style delimiters make use of a couple of characters which represents an opening and closing pair.

- Matching delimiters: `!"#$%&'*+,./:;=?@^_`|~-`
- Bracket-style delimiters: `()`, `{}`, `[]`, `<>`

Read Escaping online: https://riptutorial.com/regex/topic/4524/escaping

Chapter 10: Greedy and Lazy quantifiers

Parameters

Quantifiers	Description
`?`	Match the preceding character or subexpression 0 or 1 times (preferably 1).
`*`	Match the preceding character or subexpression 0 or more times (as many as possible).
`+`	Match the preceding character or subexpression 1 or more times (as many as possible).
`{n}`	Match the preceding character or subexpression exactly *n* times.
`{min,}`	Match the preceding character or subexpression *min* or more times (as many as possible).
`{0,max}`	Match the preceding character or subexpression *max* or fewer times (as close to *max* as possible).
`{min,max}`	Match the preceding character or subexpression at least *min* times but no more than *max* times (as close to *max* as possible).
Lazy Quantifiers	**Description**
`??`	Match the preceding character or subexpression 0 or 1 times (preferably 0).
`*?`	Match the preceding character or subexpression 0 or more times (as few as possible).
`+?`	Match the preceding character or subexpression 1 or more times (as few as possible).
`{n}?`	Match the preceding character or subexpression exactly *n* times. No difference between greedy and lazy version.
`{min,}?`	Match the preceding character or subexpression *min* or more times (as close to *min* as possible).
`{0,max}?`	Match the preceding character or subexpression *max* or fewer times (as few as possible).
`{min,max}?`	Match the preceding character or subexpression at least *min* times but no more than *max* times (as close to *min* as possible).

Remarks

Greediness

A *greedy* quantifier always attempts to repeat the sub-pattern as many times as possible before exploring shorter matches by backtracking.

Generally, a greedy pattern will match the longest possible string.

By default, all quantifiers are greedy.

Laziness

A *lazy* (also called *non-greedy* or *reluctant*) quantifier always attempts to repeat the sub-pattern as *few* times as possible, before exploring longer matches by expansion.

Generally, a lazy pattern will match the shortest possible string.

To make quantifiers lazy, just append `?` to the existing quantifier, e.g. `+?`, `{0,5}?`.

Concept of greediness and laziness only exists in backtracking engines

The notion of greedy/lazy quantifier only exists in backtracking regex engines. In non-backtracking regex engines or POSIX-compliant regex engines, quantifiers only specify the upper bound and lower bound of the repetition, without specifying how to find the match -- those engines will always match the left-most longest string regardless.

Examples

Greediness versus Laziness

Given the following input:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
```

We will use two patterns: one greedy: `A.*Z`, and one lazy: `A.*?Z`. These patterns yield the following matches:

- `A.*Z` yields 1 match: `AlazyZgreeedyAlaaazyZ` (examples: Regex101, Rubular)
- `A.*?Z` yields 2 matches: `AlazyZ` and `AlaaazyZ` (examples: Regex101, Rubular)

First focus on what `A.*Z` does. When it matched the first `A`, the `.*`, being greedy, then tries to match as many `.` as possible.

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_____________________________/
      A.* matched, Z can't match
```

Since the `Z` doesn't match, the engine backtracks, and `.*` must then match one fewer `.`:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________________/
      A.* matched, Z can't match
```

This happens a few more times, until it finally comes to this:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____________________/
      A.* matched, Z can now match
```

Now `Z` can match, so the overall pattern matches:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \_____________________/
      A.*Z matched
```

By contrast, the reluctant (lazy) repetition in `A.*?Z` first matches as few `.` as possible, and then taking more `.` as necessary. This explains why it finds two matches in the input.

Here's a visual representation of what the two patterns matched:

```
aaaaaAlazyZgreeedyAlaaazyZaaaaa
     \____/l      \______/l      l = lazy
     \_________g_________/       g = greedy
```

Example based on answer made by polygenelubricants.

The POSIX standard does not include the `?` operator, so many POSIX regex engines do not have lazy matching. While refactoring, especially with the "greatest trick ever", may help match in some cases, the only way to have true lazy matching is to use an engine that supports it.

Boundaries with multiple matches

When you have an input with well defined boundaries and are expecting more than one match in your string, you have two options:

- Using lazy quantifiers;
- Using a negated character class.

Consider the following:

You have a simple templating engine, you want to replace substrings like `$[foo]` where `foo` can be any string. You want to replace this substring with whatever based on the part between the `[]`.

You can try something like `\$\[(.*)\]`, and then use the first capture group.

The problem with this is if you have a string like `something $[foo] lalala $[bar] something else` your match will be

```
something $[foo] lalala $[bar] something else
          | \_______CG1_______/|
          \________Match_______/
```

The capture group being `foo] lalala $[bar` which may or may not be valid.

You have two solutions

1. Using laziness: In this case making `*` lazy is one way to go about finding the right things. So you change your expression to `\$\[(.*?)\]`
2. Using negated character class : `[^\]]` you change your expression to `\$\[([^\]]*)\]`.

In both solutions, the result will be the same:

```
something $[foo] lalala $[bar] something else
          | \_/|        | \_/|
          \____/        \____/
```

With the capture group being respectively `foo` and `bar`.

Using negated character class reduces backtracking issue and may save your CPU a lot of time when it comes to large inputs.

Read Greedy and Lazy quantifiers online: https://riptutorial.com/regex/topic/429/greedy-and-lazy-quantifiers

Chapter 11: Lookahead and Lookbehind

Syntax

- **Positive lookahead:** `(?=pattern)`
- **Negative lookahead:** `(?!pattern)`
- **Positive lookbehind**: `(?<=pattern)`
- **Negative lookbehind**: `(?<!pattern)`

Remarks

Not supported by all regex engines.

Additionally, many regex engines limit the patterns inside lookbehinds to fixed-length strings. For example the pattern `(?<=a+)b` should match the `b` in `aaab` but throws an error in Python.

Capturing groups are allowed and work as expected, including backreferences. The lookahead/lookbehind itself is not a capturing group, however.

Examples

Basics

A **positive lookahead** `(?=123)` asserts the text is followed by the given pattern, without including the pattern in the match. Similarly, a **positive lookbehind** `(?<=123)` asserts the text is preceded by the given pattern. Replacing the `=` with `!` negates the assertion.

Input: `123456`

- `123(?=456)` matches `123` (*positive lookahead*)
- `(?<=123)456` matches `456` (*positive lookbehind*)
- `123(?!456)` fails (*negative lookahead*)
- `(?<!123)456` fails (*negative lookbehind*)

Input: `456`

- `123(?=456)` fails
- `(?<=123)456` fails
- `123(?!456)` fails
- `(?<!123)456` matches `456`

Using lookbehind to test endings

A lookbehind can be used at the end of a pattern to ensure it ends or not in a certain way.

`([a-z ]+|[A-Z ]+)(?<! )` matches sequences of only lowercase or only uppercase words while excluding trailing whitespace.

Simulating variable-length lookbehind with \K

Some regex flavors (Perl, PCRE, Oniguruma, Boost) only support fixed-length lookbehinds, but offer the `\K` feature, which can be used to simulate variable-length lookbehind at the start of a pattern. Upon encountering a `\K`, the matched text up to this point is discarded, and only the text matching the part of the pattern *following* `\K` is kept in the final result.

```
ab+\Kc
```

Is equivalent to:

```
(?<=ab+)c
```

In general, a pattern of the form:

```
(subpattern A)\K(subpattern B)
```

Ends up being similar to:

```
(?<=subpattern A)(subpattern B)
```

Except when the B subpattern can match the same text as the A subpattern - you could end up with subtly different results, because the A subpattern still consumes the text, unlike a true lookbehind.

Read Lookahead and Lookbehind online: https://riptutorial.com/regex/topic/639/lookahead-and-lookbehind

Chapter 12: Match Reset: \K

Remarks

Regex101 defines \K functionality as:

> `\K` resets the starting point of the reported match. Any previously consumed characters are no longer included in the final match

The `\K` escape sequence is supported by several engines, languages or tools, such as:

- boost (since ???)
- grep -P ← *uses PCRE*
- Oniguruma (since 5.13.3)
- PCRE (since 7.2)
- Perl (since 5.10.0)
- PHP (since 5.2.4)
- Ruby (since 2.0.0)

...and (so far) not supported by:

- .NET
- awk
- bash
- GNU
- ICU
- Java
- Javascript
- Notepad++
- Objective-C
- POSIX
- Python
- Qt/QRegExp
- sed
- Tcl
- vim
- XML
- XPath

Examples

Search and replace using \K operator

Given the text:

foo: bar

I would like to replace anything following "foo: " with "baz", but I want to keep "foo: ". This could be done with a capturing group like this:

```
s/(foo: ).*/$1baz/
```

Which results in the text:

foo: baz

Example 1

or we could use `\K`, which "forgets" all that it has previously matched, with a pattern like this:

```
s/foo: \K.*/baz/
```

The regex matches "foo: " and then encounters the `\K`, the previously match characters are taken for granted and left by the regex meaning that only the string matched by `.*` will be replaced by "baz", resulting in the text:

foo: baz

Example 2

Read Match Reset: \K online: https://riptutorial.com/regex/topic/1338/match-reset---k

Chapter 13: Matching Simple Patterns

Examples

Match a single digit character using [0-9] or \d (Java)

`[0-9]` and `\d` are equivalent patterns (unless your Regex engine is unicode-aware and `\d` also matches things like ②). They will both match a single digit character so you can use whichever notation you find more readable.

Create a string of the pattern you wish to match. If using the \d notation, you will need to add a second backslash to escape the first backslash.

```
String pattern = "\\d";
```

Create a Pattern object. Pass the pattern string into the compile() method.

```
Pattern p = Pattern.compile(pattern);
```

Create a Matcher object. Pass the string you are looking to find the pattern in to the matcher() method. Check to see if the pattern is found.

```
Matcher m1 = p.matcher("0");
m1.matches(); //will return true

Matcher m2 = p.matcher("5");
m2.matches(); //will return true

Matcher m3 = p.matcher("12345");
m3.matches(); //will return false since your pattern is only for a single integer
```

Matching various numbers

`[a-b]` where a and b are digits in the range `0` to `9`

```
[3-7] will match a single digit in the range 3 to 7.
```

Matching multiple digits

```
\d\d       will match 2 consecutive digits
\d+        will match 1 or more consecutive digits
\d*        will match 0 or more consecutive digits
\d{3}      will match 3 consecutive digits
\d{3,6}    will match 3 to 6 consecutive digits
\d{3,}     will match 3 or more consecutive digits
```

The `\d` in the above examples can be replaced with a number range:

```
[3-7][3-7]    will match 2 consecutive digits that are in the range 3 to 7
[3-7]+        will match 1 or more consecutive digits that are in the range 3 to 7
[3-7]*        will match 0 or more consecutive digits that are in the range 3 to 7
[3-7]{3}      will match 3 consecutive digits that are in the range 3 to 7
[3-7]{3,6}    will match 3 to 6 consecutive digits that are in the range 3 to 7
[3-7]{3,}     will match 3 or more consecutive digits that are in the range 3 to 7
```

You can also select specific digits:

```
[13579]       will only match "odd" digits
[02468]       will only match "even" digits
1|3|5|7|9     another way of matching "odd" digits - the | symbol means OR
```

Matching numbers in ranges that contain more than one digit:

```
\d|10        matches 0 to 10    single digit OR 10.  The | symbol means OR
[1-9]|10     matches 1 to 10    digit in range 1 to 9 OR 10
[1-9]|1[0-5] matches 1 to 15    digit in range 1 to 9 OR 1 followed by digit 1 to 5
\d{1,2}|100  matches 0 to 100   one to two digits OR 100
```

Matching numbers that divide by other numbers:

```
\d*0         matches any number that divides by 10  - any number ending in 0
\d*00        matches any number that divides by 100 - any number ending in 00
\d*[05]      matches any number that divides by 5   - any number ending in 0 or 5
\d*[02468]   matches any number that divides by 2   - any number ending in 0,2,4,6 or 8
```

matching numbers that divide by 4 - any number that is 0, 4 or 8 or ends in 00, 04, 08, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84, 88, 92 or 96

```
[048]|\d*(00|04|08|12|16|20|24|28|32|36|40|44|48|52|56|60|64|68|72|76|80|84|88|92|96)
```

This can be shortened. For example, instead of using `20|24|28` we can use `2[048]`. Also, as the 40s, 60s and 80s have the same pattern we can include them: `[02468][048]` and the others have a pattern too `[13579][26]`. So the whole sequence can be reduce to:

```
[048]|\d*([02468][048]|[13579][26])    - numbers divisible by 4
```

Matching numbers that don't have a pattern like those divisible by 2,4,5,10 etc can't always be done succinctly and you usually have to resort to a range of numbers. For example matching all numbers that divide by 7 within the range of 1 to 50 can be done simple by listing all those numbers:

```
7|14|21|28|35|42|49

or you could do it this way

7|14|2[18]|35|4[29]
```

Matching leading/trailing whitespace

Trailing spaces

`\s*$`: This will match any (`*`) whitespace (`\s`) at the end (`$`) of the text

Leading spaces

`^\s*`: This will match any (`*`) whitespace (`\s`) at the beginning (`^`) of the text

Remarks

`\s` is a common metacharacter for several RegExp engines, and is meant to capture whitespace characters (spaces, newlines and tabs for example). **Note**: it probably *won't* capture all the unicode space characters. Check your engines documentation to be sure about this.

Match any float

```
[\+\-]?\d+(\.\d*)?
```

This will match any signed float, if you don't want signs or are parsing an equation remove `[\+\-]?` so you have `\d+(\.\d+)?`

Explanation:

- `\d+` matches any integer
- `()?` means the contents of the parentheses are optional but always have to appear together
- '\.' matches '.', we have to escape this since '.' normally matches any character

So this expression will match

```
5
+5
-5
5.5
+5.5
-5.5
```

Selecting a certain line from a list based on a word in certain location

I have the following list:

```
1. Alon Cohen
2. Elad Yaron
3. Yaron Amrani
4. Yogev Yaron
```

I want to select the first name of the guys with the Yaron surname.

Since I don't care about what number it is I'll just put it as whatever digit it is and a matching dot and space after it from the beginning of the line, like this: `^[\d]+\.\s`.

Now we'll have to match the space and the first name, since we can't tell whether it's capital or small letters we'll just match both: `[a-zA-Z]+\s` or `[a-Z]+\s` and can also be `[\w]+\s`.

Now we'll specify the required surname to get only the lines containing Yaron as a surname (at the end of the line): `\sYaron$`.

Putting this all together `^[\d]+\.\s[\w]+\sYaron$`.

Live example: https://regex101.com/r/nW4fH8/1

Read Matching Simple Patterns online: https://riptutorial.com/regex/topic/343/matching-simple-patterns

Chapter 14: Named capture groups

Syntax

- Build a named capture group (`X` being the pattern you want to capture):

  (?'name'X) (?X) (?PX)

- Reference a named capture group:

  ${name} \{name} g\{name}

Remarks

Python and Java don't allow multiple groups to use the same name.

Examples

What a named capture group looks like

Given the flavors, the named capture group may looks like this:

```
(?'name'X)
(?<name>X)
(?P<name>X)
```

With `X` being the pattern you want to capture. Let's consider the following string:

> Once upon a time there was a *pretty little girl*...
>
> Once upon a time there was a *unicorn with an hat*...
>
> Once upon a time there was a *boat with a pirate flag*...

In which I want to capture the subject (in *italic)* of every lines. I'll use the following expression `.* was a (?<subject>[\w ]+)[.]{3}.`

The matching result will hold:

```
MATCH 1
subject    [29-47]    `pretty little girl`
MATCH 2
subject    [80-99]    `unicorn with an hat`
MATCH 3
subject    [132-155]    `boat with a pirate flag`
```

Reference a named capture group

As you may (or not) know, you can reference a capture group with:

```
$1
```

`1` being the group number.

In the same way, you can reference a named capture group with:

```
${name}
\{name}
g\{name}
```

Let's take the preceding example and replace the matches with

```
The hero of the story is a ${subject}.
```

The result we will obtain is:

```
The hero of the story is a pretty little girl.
The hero of the story is a unicorn with an hat.
The hero of the story is a boat with a pirate flag.
```

Read Named capture groups online: https://riptutorial.com/regex/topic/744/named-capture-groups

Chapter 15: Password validation regex

Examples

A password containing at least 1 uppercase, 1 lowercase, 1 digit, 1 special character and have a length of at least of 10

As the characters/digits can be anywhere within the string, we require lookaheads. Lookaheads are of `zero width` meaning they do not consume any string. In simple words the position of checking resets to the original position after each condition of lookahead is met.

Assumption :- *Considering non-word characters as special*

```
^(?=.{10,}$)(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*\W).*$
```

Before proceeding to explanation, let's take a look how the regex `^(?=.*[a-z])` works (*length is not considered here*) on string `1$d%aA`

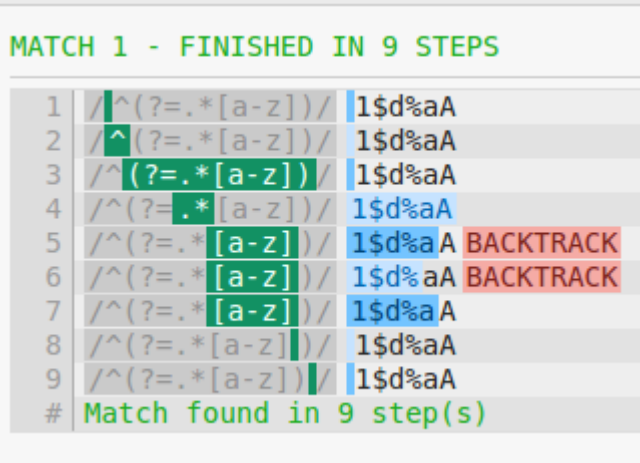


Image Credit :- **https://regex101.com/**

Things to notice

- Checking is started from beginning of the string due to anchor tag `^`.
- The position of checking is being reset to the starting after condition of lookahead is met.

Regex Breakdown

```
^ #Starting of string
 (?=.{10,}$) #Check there is at least 10 characters in the string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[a-z]) #Check if there is at least one lowercase in string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[A-Z]) #Check if there is at least one uppercase in string.
             #As this is lookahead the position of checking will reset to starting again
 (?=.*[0-9]) #Check if there is at least one digit in string.
             #As this is lookahead the position of checking will reset to starting again
```

```
 (?=.*\W) #Check if there is at least one special character in string.
          #As this is lookahead the position of checking will reset to starting again
.*$ #Capture the entire string if all the condition of lookahead is met. This is not required
if only validation is needed
```

We can also use the *non-greedy* version of the above regex

```
^(?=.{10,}$)(?=.*?[a-z])(?=.*?[A-Z])(?=.*?[0-9])(?=.*?\W).*$
```

A password containing at least 2 uppercase, 1 lowercase, 2 digits and is of length of at least 10

This can be done with a bit of modification in the above regex

```
^(?=.{10,}$)(?=(?:.*?[A-Z]){2})(?=.*?[a-z])(?=(?:.*?[0-9]){2}).*$
```

or

```
^(?=.{10,}$)(?=(?:.*[A-Z]){2})(?=.*[a-z])(?=(?:.*[0-9]){2}).*
```

Let's see how a simple regex `^(?=(?:.*?[A-Z]){2})` works on string `abcAdefD`

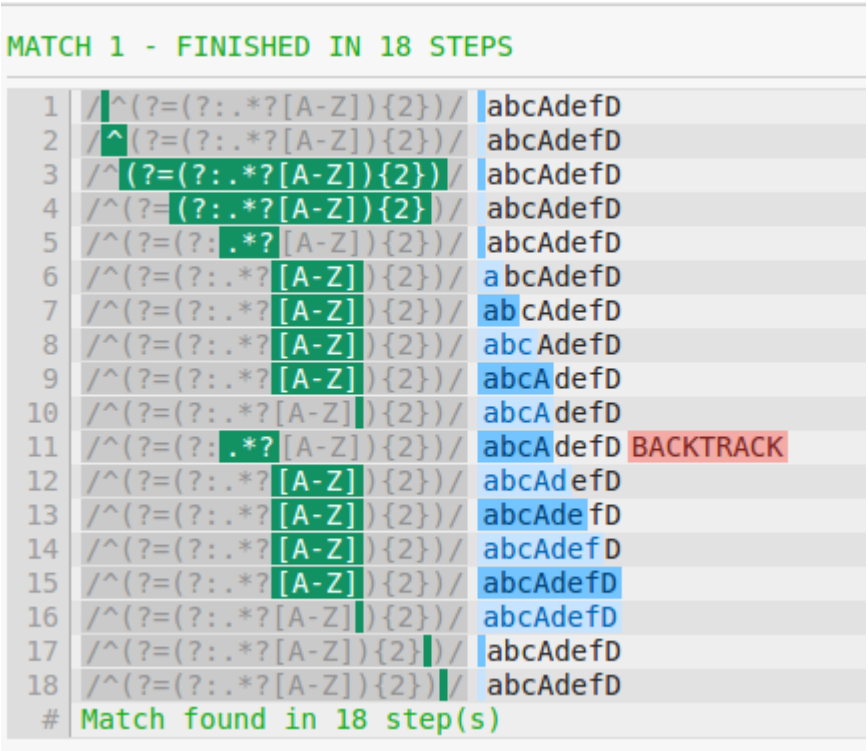


Image Credit :- **https://regex101.com/**

Read Password validation regex online: https://riptutorial.com/regex/topic/5340/password-validation-regex

Chapter 16: Possessive Quantifiers

Remarks

NB Emulating possessive quantifiers

Examples

Basic Use of Possessive Quantifiers

Possessive quantifiers are another class of quantifiers in many regex flavours that allow backtracking to, effectively, be disabled for a given token. This can help improve performance, as well as preventing matches in certain cases.

The class of possessive quantifiers can be distinguished from lazy or greedy quantifiers by the addition of a `+` after the quantifier, as seen below:

Quantifier	Greedy	Lazy	Possessive
Zero or more	`*`	`*?`	`*+`
One or more	`+`	`+?`	`++`
Zero or one	`?`	`??`	`?+`

Consider, for instance, the two patterns `".*"` and `".*+"`, operating on the string `"abc"d`. In both cases, the `"` at the beginning of the string is matched, but after that the two patterns will have different behaviours and outcomes.

The greedy quantifier will then slurp the rest of the string, `abc"d`. Because this does not match the pattern, it will then backtrack and drop the `d`, leaving the **quantifier** containing `abc"`. Because this still does not match the pattern, the quantifier will drop the `"`, leaving it containing only `abc`. This matches the pattern (as the `"` is matched by a literal, rather than the quantifier), and the regex reports success.

The possessive quantifier will also slurp the rest of the string, but, unlike the greedy quantifier, it will not backtrack. Since its contents, `abc"d`, do not permit the rest of the pattern of the match, the regex will stop and report failure to match.

Because the possessive quantifiers do not do backtracking, they can result in a significant performance increase on long or complex patterns. They can, however, be dangerous (as illustrated above) if one is not aware of how, precisely, quantifiers work internally.

Read Possessive Quantifiers online: https://riptutorial.com/regex/topic/5916/possessive-quantifiers

Chapter 17: Recursion

Remarks

Recursion is mostly available in Perl-compatible flavors, such as:

- Perl
- PCRE
- Oniguruma
- Boost

Examples

Recurse the whole pattern

The construct `(?R)` is equivalent to `(?0)` (or `\g<0>`) - it lets you recurse the whole pattern:

```
<(?>[^<>]+|(?R))+>
```

This will match properly balanced angle brackets with any text in-between the brackets, like `<a<b>c<d>e>`.

Recurse into a subpattern

You can recurse into a subpattern using the following constructs (depending on the flavor), assuming `n` is a capturing group number, and `name` the name of a capturing group.

- `(?n)`
- `\g<n>`
- `\g'0'`
- `(?&name)`
- `\g<name>`
- `\g'name'`
- `(?P>name)`

The following pattern:

```
\[(?<angle><(?&angle)*+>)*\]
```

Will match text such as: `[<<><>><>]` - well balanced angle brackets within square brackets. Recursion is often used for balanced constructs matching.

Subpattern definitions

The `(?(DEFINE)...)` construct lets you define subpatterns you may reference later through recursion. When encountered in the pattern it will *not* be matched against.

This group should contain named subpattern definitions, which will be accessible only through recursion. You can define grammars this way:

```
(?x) # ignore pattern whitespace
(?(DEFINE)
  (?<string> ".*?" )
  (?<number> \d+ )
  (?<value>
    \s* (?:
        (?&string)
      | (?&number)
      | (?&list)
    ) \s*
  )
  (?<list> \[ (?&value) (?: , (?&value) )* \] )
)
^(?&value)$
```

This pattern will validate text like the following:

```
[42, "abc", ["foo", "bar"], 10]
```

Note how a list can contain one or more values, and a value can itself be a list.

Relative group references

Subpatterns can be referenced with their *relative* group number:

- `(?-1)` will recurse into the *previous* group
- `(?+1)` will recurse into the *next* group

Also usable with the `\g<N>` syntax.

Backreferences in recursions (PCRE)

In PCRE, matched groups used for backreferences before a recursion are kept in the recursion. But after the recursion they all reset to what they were before entering it. In other words, matched groups in the recursion are all forgotten.

For example:

```
(?J)(?(DEFINE)(\g{a}(?<a>b)\g{a}))(?<a>a)\g{a}(?1)\g{a}
```

matches

```
aaabba
```

Recursions are atomic (PCRE)

In PCRE, it doesn't trackback after the first match for a recursion is found. So

```
(?(DEFINE)(aaa|aa|a))(?1)ab
```

doesn't match

```
aab
```

because after it matched `aa` in the recursion, it never try again to match only `a`.

Read Recursion online: https://riptutorial.com/regex/topic/739/recursion

Chapter 18: Regex modifiers (flags)

Introduction

Regular expression patterns are often used with *modifiers* (also called *flags*) that redefine regex behavior. Regex modifiers can be *regular* (e.g. `/abc/i`) and *inline* (or *embedded*) (e.g. `(?i)abc`). The most common modifiers are global, case-insensitive, multiline and dotall modifiers. However, regex flavors differ in the number of supported regex modifiers and their types.

Remarks

PCRE Modifiers

Modifier	Inline	Description
PCRE_CASELESS	(?i)	Case insensitive match
PCRE_MULTILINE	(?m)	Multiple line matching
PCRE_DOTALL	(?s)	. matches new lines
PCRE_ANCHORED	(?A)	Meta-character ^ matches only at the start
PCRE_EXTENDED	(?x)	White-spaces are ignored
PCRE_DOLLAR_ENDONLY	n/a	Meta-character $ matches only at the end
PCRE_EXTRA	(?X)	Strict escape parsing
PCRE_UTF8		Handles `UTF-8` characters
PCRE_UTF16		Handles `UTF-16` characters
PCRE_UTF32		Handles `UTF-32` characters
PCRE_UNGREEDY	(?U)	Sets the engine to lazy matching
PCRE_NO_AUTO_CAPTURE	(?:)	Disables auto-capturing groups

Java Modifiers

Modifier (`Pattern.###`)	Value	Description
UNIX_LINES	1	Enables Unix lines mode.

Modifier (`Pattern.###`)	Value	Description
CASE_INSENSITIVE	2	Enables case-insensitive matching.
COMMENTS	4	Permits whitespace and comments in a pattern.
MULTILINE	8	Enables multiline mode.
LITERAL	16	Enables literal parsing of the pattern.
DOTALL	32	Enables dotall mode.
UNICODE_CASE	64	Enables Unicode-aware case folding.
CANON_EQ	128	Enables canonical equivalence.
UNICODE_CHARACTER_CLASS	256	Enables the Unicode version of Predefined character classes and POSIX character classes.

Examples

DOTALL modifier

A regex pattern where a DOTALL modifier (in most regex flavors expressed with `s`) changes the behavior of `.` enabling it to match a newline (LF) symbol:

```
/cat (.*?) dog/s
```

This Perl-style regex will match a string like `"cat fled from\na dog"` capturing `"fled from\na"` into Group 1.

An inline version: `(?s)` (e.g. `(?s)cat (.*?) dog`)

Note: In Ruby, the DOTALL modifier equivalent is `m`, `Regexp::MULTILINE` modifier (e.g. `/a.*b/m`).

Note: JavaScript does not provide a DOTALL modifier, so a `.` can never be allowed to match a newline character. In order to achieve the same effect, a workaround is necessary, e. g. substituting all the `.`s with a catch-all character class like `[\S\s]`, or a *not nothing* character class `[^]` (however, this construct will be treated as an error by all other engines, and is thus not portable).

MULTILINE modifier

Another example is a MULTILINE modifier (usually expressed with `m` flag (not in Oniguruma (e.g. Ruby) that uses `m` to denote a DOTALL modifier)) that makes `^` and `$` anchors match the start/end of a *line*, not the start/end of the whole string.

```
/^My Line \d+$/gm
```

will find all *lines* that start with `My Line`, then contain a space and 1+ digits up to the line end.

An inline version: `(?m)` (e.g. `(?m)^My Line \d+$`)

NOTE: In Oniguruma (e.g. in Ruby), and also in almost any text editors supporting regexps, the `^` and `$` anchors denote *line* start/end positions *by default*. You need to use `\A` to define the whole document/string start and `\z` to denote the document/string end. The difference between the `\Z` and `\z` is that the former can match before the final newline (LF) symbol at the end of the string (e.g. `/\Astring\Z/` will find a match in `"string\n"`) (except Python, where `\Z` behavior is equal to `\z` and `\z` anchor is not supported).

IGNORE CASE modifier

The common modifier to ignore case is `i`:

```
/fog/i
```

will match `Fog`, `foG`, etc.

The inline version of the modifier looks like `(?i)`.

Notes:

In Java, by default, *case-insensitive matching assumes that only characters in the US-ASCII charset are being matched. Unicode-aware case-insensitive matching can be enabled by specifying the `UNICODE_CASE` flag in conjunction with this (`CASE_INSENSITIVE`) flag*. (e.g. `Pattern p = Pattern.compile("YOUR_REGEX", Pattern.CASE_INSENSITIVE | Pattern.UNICODE_CASE);`). Some more on this can be found at *Case-Insensitive Matching in Java RegEx*. Also, `UNICODE_CHARACTER_CLASS` can be used to make matching Unicode aware.

VERBOSE / COMMENT / IgnorePatternWhitespace modifier

The modifier that allows using whitespace inside some parts of the pattern to format it for better readability and to allow comments starting with `#`:

```
/(?x)^          # start of string
  (?=\D*\d)     # the string should contain at least 1 digit
  (?!\d+$)      # the string cannot consist of digits only
  \#            # the string starts with a hash symbol
  [a-zA-Z0-9]+ # the string should have 1 or more alphanumeric symbols
  $             # end of string
/
```

Example of a string: `#word1here`. Note the `#` symbol is escaped to denote a literal `#` that is part of a pattern.

Unescaped white space in the regular expression pattern is ignored, escape it to make it a part of the pattern.

Usually, the whitespace inside character classes (`[...]`) is treated as a literal whitespace, except

in Java.

Also, it is worth mentioning that in PCRE, .NET, Python, Ruby Oniguruma, ICU, Boost regex flavors one can use `(?#:...)` comments inside the regex pattern.

Explicit Capture modifier

This is a .NET regex specific modifier expressed with `n`. When used, unnamed groups (like `(\d+)`) are not captured. Only valid captures are explicitly named groups (e.g. `(?<name> subexpression)`).

```
(?n)(\d+)-(\w+)-(?<id>\w+)
```

will match the whole `123-1_abc-00098`, but `(\d+)` and `(\w+)` won't create groups in the resulting match object. The only group will be `${id}`. See demo.

UNICODE modifier

The UNICODE modifier, usually expressed as `u` (PHP, Python) or `U` (Java), makes the regex engine treat the pattern and the input string as Unicode strings and patterns, make the pattern shorthand classes like `\w`, `\d`, `\s`, etc. Unicode-aware.

```
/\A\p{L}+\z/u
```

is a PHP regex to match strings that consist of 1 or more Unicode letters. See the regex demo.

Note that in **PHP**, the `/u` modifier enables the PCRE engine to handle strings as UTF8 strings (by turning on `PCRE_UTF8` verb) and make the shorthand character classes in the pattern Unicode aware (by enabling `PCRE_UCP` verb, see more at *pcre.org*).

> **Pattern and subject strings are treated as UTF-8.** This modifier is available from PHP 4.1.0 or greater on Unix and from PHP 4.2.3 on win32. UTF-8 validity of the pattern and the subject is checked since PHP 4.3.5. An invalid subject will cause the preg_* function to match nothing; an invalid pattern will trigger an error of level E_WARNING. Five and six octet UTF-8 sequences are regarded as invalid since PHP 5.3.4 (resp. PCRE 7.3 2007-08-28); formerly those have been regarded as valid UTF-8.

In Python 2.x, the `re.UNICODE` only affects the pattern itself: *Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` dependent on the Unicode character properties database.*

An inline version: `(?u)` in Python, `(?U)` in Java. For example:

```
print(re.findall(ur"(?u)\w+", u"Dąb")) # [u'D\u0105b']
print(re.findall(r"\w+", u"Dąb"))      # [u'D', u'b']

System.out.println("Dąb".matches("(?U)\\w+")); // true
System.out.println("Dąb".matches("\\w+"));     // false
```

PCRE_DOLLAR_ENDONLY modifier

The PCRE-compliant *PCRE_DOLLAR_ENDONLY* modifier that makes the `$` anchor match at the *very end of the string* (excluding the position before the final newline in the string).

```
/^\d+$/D
```

is equal to

```
/^\d+\z/
```

and matches a whole string that consists of 1 or more digits and will not match `"123\n"`, but will match `"123"`.

PCRE_ANCHORED modifier

Another PCRE-compliant modifier expressed with `/A` modifier. If this modifier is set, the pattern is forced to be "anchored", that is, it is constrained to match only at the start of the string which is being searched (the "subject string"). This effect can also be achieved by appropriate constructs in the pattern itself, which is the only way to do it in Perl.

```
/man/A
```

is the same as

```
/^man/
```

PCRE_UNGREEDY modifier

The PCRE-compliant PCRE_UNGREEDY flag expressed with `/U`. It switches greediness inside a pattern: `/a.*?b/U` = `/a.*b/` and vice versa.

PCRE_INFO_JCHANGED modifier

One more PCRE modifier that allows the use of duplicate named groups.

NOTE: only *inline* version is supported - `(?J)`, and must be placed at the start of the pattern.

If you use

```
/(?J)\w+-(?:new-(?<val>\w+)|\d+-empty-(?<val>[^-]+)-collection)/
```

the "val" group values will be never empty (will always be set). A similar effect can be achieved with branch reset though.

PCRE_EXTRA modifier

A PCRE modifier that causes an error if any backslash in a pattern is followed by a letter that has no special meaning. By default, a backslash followed by a letter with no special meaning is treated as a literal.

E.g.

```
/big\y/
```

will match `bigy`, but

```
/big\y/X
```

will throw an exception.

Inline version: `(?X)`

Read Regex modifiers (flags) online: https://riptutorial.com/regex/topic/5138/regex-modifiers--flags-

Chapter 19: Regex Pitfalls

Examples

Why doesn't dot (.) match the newline character ("\n")?

`.*` in regex basically means "catch **everything** until the end of input".

So, for simple strings, like `hello world`, `.*` works perfectly. But if you have a string representing, for example, lines in a file, these lines would be separated by a *line separator*, such as `\n` (newline) on Unix-like systems and `\r\n` (carriage return and newline) on Windows.

By default in most regex engines, `.` **doesn't** match newline characters, so the matching stops at the end of each *logical line*. If you want `.` to match **really** everything, including newlines, you need to enable "dot-matches-all" mode in your regex engine of choice (for example, add `re.DOTALL` flag in Python, or `/s` in PCRE.

Why does a regex skip some closing brackets/parentheses and match them afterwards?

Consider this example:

> He went into the cafe "Dostoevski" and said: "Good evening."

Here we have two sets of quotes. Let's assume we want to match both, so that our regex matches at `"Dostoevski"` **and** `"Good evening."`

At first, you could be tempted to keep it simple:

```
".*"  # matches a quote, then any characters until the next quote
```

But it doesn't work: it matches from the first quote in `"Dostoevski"` and **until** the closing quote in `"Good evening."`, including the `and said:` part. Regex101 demo

Why did it happen?

This happens because the regex engine, when it encounters `.*`, "eats up" all of the input to the very end. Then, it needs to match the final `"`. So, it "backs off" from the end of the match, letting go of the matched text until the first `"` is found - and it is, of course, the last `"` in the match, at the end of `"Good evening."` part.

How to prevent this and match exactly to the first quotes?

Use `[^"]*`. It doesn't eat all the input - only until the first `"`, just as needed. Regex101 demo

Read Regex Pitfalls online: https://riptutorial.com/regex/topic/10747/regex-pitfalls

Chapter 20: Regular Expression Engine Types

Examples

NFA

A NFA (Nondeterministic Finite Automaton) engine is *driven by the pattern*.

Principle

The regex pattern is parsed into a tree.

The *current position* pointer is set to the start of the input string, and a match is attempted at this position. If the match fais, the position is incremented to the next character in the string and another match is attempted from this position. This process is repeated until a match is found or the end of the input string is reached.

For each match attempt

The algorithm works by performing a traversal of the pattern tree for a given starting position. As it progresses through the tree, it updates the *current input position* by consuming matching characters.

If the algorithm encounters a tree node which does not match the input string at the current position, it will have to *backtrack*. This is performed by going back to the parent node in the tree, resetting the current input position to the value it had upon entering the parent node, and trying the next alternative branch.

If the algorithm manages to exit the tree, it reports a successful match. Otherwise, when all possibilities have been tried, the match fails.

Optimizations

Regex engines usually apply some optimizations for better performance. For instance, if they determine that a match must start with a given character, they will attempt a match only at those positions in the input string where that character appears.

Example

Match `a(b|c)a` against the input string `abeacab`:

The pattern tree could looks something like:

```
CONCATENATION
    EXACT: a
    ALTERNATION
        EXACT: b
        EXACT: c
    EXACT: a
```

The match processes as follows:

```
a(b|c)a      abeacab
^            ^
```

`a` is found in the input string, consume it and proceed to the next item in the pattern tree: the alternation. Try the first possibility: an exact `b`.

```
a(b|c)a      abeacab
  ^           ^
```

`b` is found, so the alternation succeeds, consume it and proceed to the next item in the concatenation: an exact `a`:

```
a(b|c)a      abeacab
      ^        ^
```

`a` is *not* found at the expected position. Backtrack to the alternation, reset the input position to the value it had upon entering the alternation for the first time, and try the *second* alternative:

```
a(b|c)a      abeacab
    ^         ^
```

`c` is *not* found at this position. Backtrack to the concatenation. There are no other possibilities to try at this point, so there is no match at the start of the string.

Attempt a second match at the next input position:

```
a(b|c)a      abeacab
^             ^
```

`a` does *not* match there. Attempt another match at the next position:

```
a(b|c)a      abeacab
^              ^
```

No luck either. Advance to the next position.

```
a(b|c)a      abeacab
^               ^
```

`a` matches, so consume it and enter the alternation:

```
a(b|c)a      abeacab
  ^              ^
```

`b` does not match. Attempt the second alternative:

```
a(b|c)a      abeacab
    ^              ^
```

`c` matches, so consume it and advance to the next item in the concatenation:

```
a(b|c)a      abeacab
      ^             ^
```

`a` matches, and the end of the tree has been reached. Report a successful match:

```
a(b|c)a      abeacab
               \_/
```

DFA

A DFA (Deterministic Finite Automaton) engine is *driven by the input*.

Principle

The algorithm scans through the input string *once*, and remembers all possible paths in the regex which could match. For instance, when an alternation is encountered in the pattern, two new paths are created and attempted independently. When a given path does not match, it is dropped from the possibilities set.

Implications

The matching time is bounded by the input string size. There is no backtracking, and the engine can find multiple matches simultaneously, even overlapping matches.

The main drawback of this method is the reduced feature set which can be supported by the engine, compared to the NFA engine type.

Example

Match `a(b|c)a` against `abadaca`:

```
abadaca      a(b|c)a
^            ^       Attempt 1      ==> CONTINUE

abadaca      a(b|c)a
 ^           ^       Attempt 2      ==> FAIL
               ^     Attempt 1.1    ==> CONTINUE
                  ^  Attempt 1.2    ==> FAIL

abadaca      a(b|c)a
  ^          ^       Attempt 3      ==> CONTINUE
                  ^  Attempt 1.1    ==> MATCH

abadaca      a(b|c)a
   ^         ^       Attempt 4      ==> FAIL
               ^     Attempt 3.1    ==> FAIL
                  ^  Attempt 3.2    ==> FAIL

abadaca      a(b|c)a
    ^        ^       Attempt 5      ==> CONTINUE

abadaca      a(b|c)a
     ^       ^       Attempt 6      ==> FAIL
               ^     Attempt 5.1    ==> FAIL
                  ^  Attempt 5.2    ==> CONTINUE

abadaca      a(b|c)a
      ^      ^       Attempt 7      ==> CONTINUE
                  ^  Attempt 5.2    ==> MATCH

abadaca      a(b|c)a
       ^       ^     Attempt 7.1    ==> FAIL
                  ^  Attempt 7.2    ==> FAIL
```

Read Regular Expression Engine Types online: https://riptutorial.com/regex/topic/2861/regular-expression-engine-types

Chapter 21: Substitutions with Regular Expressions

Parameters

Inline	Description
$number	Substitutes the substring matched by group number.
${name}	Substitutes the substring matched by a named group name.
$$	Escaped '$' character in the result (replacement) string.
$& *($0)*	Replaces with the whole matched string.
$+ *($&)*	Substitutes the matched text to the last group captured.
$`	Substitutes all the matched text with every non-matched text before the match.
$'	Substitutes all the matched text with every non-matched text after the match.
$_	Substitutes all the matched text to the entire string.
Note:	*Italic* terms means the strings are volatile (May vary depending on your regex flavor).

Examples

Basics of Substitution

One of the most common and useful ways to replace text with regex is by using Capture Groups. Or even a Named Capture Group, as a reference to store, or replace the data.

There are two terms pretty look alike in regex's docs, so it may be important to never mix-up **Substitutions** (i.e. `$1`) with **Backreferences** (i.e. `\1`). Substitution terms are used in a replacement text; Backreferences, in the pure Regex expression. Even though some programming languages accept both for substitutions, it's not encouraging.

Let's we say we have this regex: `/hello(\s+)world/i`. Whenever `$number` is referenced (in this case, `$1`), the whitespaces matched by `\s+` will be replaced instead.
The same result will be exposed with the regex: `/hello(?<spaces>\s+)world/i`. And as we have a named group here, we can also use `${spaces}`.

In this same example, we can also use `$0` or `$&` (**Note:** `$&` may be used as `$+` instead, meaning to retrieve the **LAST** capture group in other regex engines), depending on the regex flavor you're

working with, to get the whole matched text. (i.e. `$&` shall return `hEllo woRld` for the string: `hEllo woRld of Regex!`)

Take a look at this simple example of substitution using John Lennon's adapted quote by using the `$number` and the `${name}` syntax:

Simple capture group example:

```
/(Happy)\./g
```

Test String

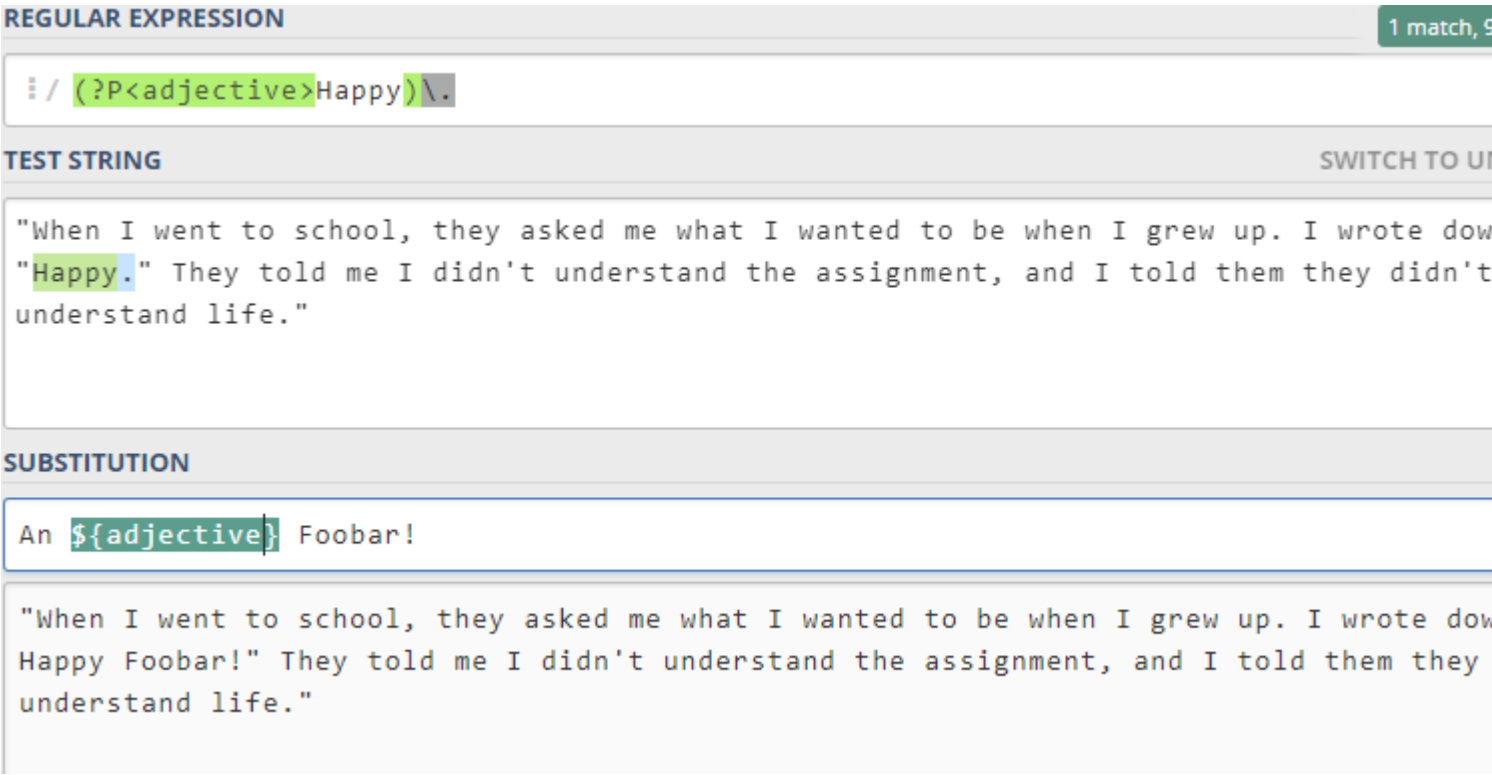
```
"When I went to school, they asked me what I wanted to be when I grew up. I w
me I didn't understand the assignment, and I told them they didn't understand
```

Substitution

```
An $1 Foobar!
```

```
"When I went to school, they asked me what I wanted to be when I grew up. I w
They told me I didn't understand the assignment, and I told them they didn't
```

Named capture group example:



Advanced Replacement

Some programming languages have its own Regex peculiarities, for example, the `$+` term (in C#, Perl, VB etc.) which replaces the matched text to the last group captured.

Example:

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//       The dog jumped over the fence.
```

Example from Microsoft Official's Developer Network [1]

Other rare substitution terms are `$`` and `$'`:

`$`` = Replaces matches to the text **before** the matching string
`$'` = Replaces matches to the text **after** the matching string

Due to this fact, these replacements strings should do their work like this:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$`"
Output: "part1part1part3" //Note that part2 was replaced with part1, due &` term
--------------------------------------------------------------------------------
Regex: /part2/
Input: "part1part2part3"
Replacement: "$'"
Output: "part1part3part3" //Note that part2 was replaced with part3, due &' term
```

Here is an example of these substitutions working on a piece of javascript:

```
var rgx = /middle/;
var text = "Your story must have a beginning, middle, and end"
console.log(text.replace(rgx, "$`"));
//Logs: "Your story must have a beginning, Your story must have a beginning, , and end"
console.log(text.replace(rgx, "$'"))
//Logs: "Your story must have a beginning, , and end, and end"
```

There is also the term `$_` which retrieves the whole matched text instead:

```
Regex: /part2/
Input: "part1part2part3"
Replacement: "$_"
Output: "part1part1part2part3part3" //Note that part2 was replaced with part1part2part3,
                                                                    // due $_ term
```

Converting this to VB would give us this:

```
Imports System.Text.RegularExpressions

Module Example
   Public Sub Main()
      Dim input As String = "ABC123DEF456"
      Dim pattern As String = "\d+"
      Dim substitution As String = "$_"
      Console.WriteLine("Original string:          {0}", input)
      Console.WriteLine("String with substitution: {0}", _
                        Regex.Replace(input, pattern, substitution))
   End Sub
End Module
' The example displays the following output:
'       Original string:          ABC123DEF456
'       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

Example from Microsoft Official's Developer Network [2]

And the last but not least substitution term is `$$`, which translated to a regex expression would be the same as `\$` (An escaped version of the literal **$**).

If you want to match a string like this: `USD: $3.99` for example, and want to store the `3.99`, but replace it as `$3.99` with only one regex, you may use:

```
Regex: /USD:\s+\$([\d.]+)/
```

```
Input: "USD: $3.99"
Replacement: "$$$1"
To Store: "$1"
Output: "$3.99"
Stored: "3.99"
```

If you want to test this with Javascript, you may use the code:

```
var rgx = /USD:\s+\$([\d.]+)/;
var text = "USD: $3.99";
var stored = parseFloat(rgx.exec(text)[1]);
console.log(stored); //Logs 3.99
console.log(text.replace(rgx, "$$$1")); //Logs $3.99
```

References

[1]: Substituting the Last Captured Group
[2]: Substituting the Entire Input String

Read Substitutions with Regular Expressions online:
https://riptutorial.com/regex/topic/9852/substitutions-with-regular-expressions

Chapter 22: Useful Regex Showcase

Examples

Match a date

You should remember that regex was designed for matching a date (or not). Saying that a date is *valid* is a much more complicated struggle, since it will require a lot of exception handling (see leap year conditions).

Let's start by matching the month (1 - 12) with an optional leading 0:

```
0?[1-9]|1[0-2]
```

To match the day, also with an optional leading 0:

```
0?[1-9]|[12][0-9]|3[01]
```

And to match the year (let's just assume the range 1900 - 2999):

```
(?:19|20)[0-9]{2}
```

The separator can be a space, a dash, a slash, empty, etc. Feel free to add anything you feel may be used as a separator:

```
[-\\/ ]?
```

Now you concatenate the whole thing and get:

```
(0?[1-9]|1[0-2])[-\\/ ]?(0?[1-9]|[12][0-9]|3[01])[-/ ]?(?:19|20)[0-9]{2} // MMDDYYYY
(0?[1-9]|[12][0-9]|3[01])[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}[-\\/ ]?(0?[1-9]|1[0-2])[-/ ]?(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

If you want to be a bit more pedantic, you can use a back reference to be sure that the two separators will be the same:

```
(0?[1-9]|1[0-2])([-\\/ ]?)(0?[1-9]|[12][0-9]|3[01])\2(?:19|20)[0-9]{2} // MMDDYYYY
                                                    ^ refer to [-/ ]
(0?[1-9]|[12][0-9]|3[01])([-\\/ ]?)(0?[1-9]|1[0-2])\2(?:19|20)[0-9]{2} // DDMMYYYY
(?:19|20)[0-9]{2}([-\\/ ]?)(0?[1-9]|1[0-2])\2(0?[1-9]|[12][0-9]|3[01]) // YYYYMMDD
```

Match an email address

> Matching an email address within a string is a hard task, because the specification defining it, the RFC2822, is complex making it hard to implement as a regex. For more details why it is not a good idea to match an email with a regex, please refer to the

> antipattern example when not to use a regex: for matching emails. The best advice to note from that page is to use a peer reviewed and widely library in your favorite language to implement this.

Validate an email address format

When you need to rapidly validate an entry to make sure it *looks like* an email, the best option is to keep it simple:

```
^\S{1,}@\S{2,}\.\S{2,}$
```

That regex will check that the mail address is a non-space separated sequence of characters of length greater than one, followed by an `@`, followed by two sequences of non-spaces characters of length two or more separated by a `.`. It's not perfect, and might validate invalid addresses (according to the format), but most importantly, it's not invalidating valid addresses.

Check the address exists

The only reliable way to check that an email is valid is to check for its existence. There used to be the `VRFY` SMTP command that has been designed for that purpose, but sadly, after being abused by spammers it's now not available anymore.

So the only way you're left with to check that the mail is valid and exists is to actually send an e-mail to that address.

Huge Regex alternatives

Though, it's not impossible to validate an address email using a regex. The only issues is that the closer to the specification those regexes will be, the bigger they will be and as a consequency they are impossibly hard to read and maintain. Below, you'll find example of such more accurate regex that are being used in some libraries.

> The following regex are given for documentation and learning purposes, copy pasting them in your code is a bad idea. Instead, use that library directly, so you can rely on upstream code and peer developers to keep your email parsing code up to date and maintained.

Perl Address matching module

The best examples of such regex are in some languages standard libraries. For example, there's one from the `RFC::RFC822::Address` module in the Perl library that tries to be as accurate as possible according to the RFC. For your curiosity you can find a version of that regex at this URL, that has been generated from the grammar, and if you're tempted to copy paste it, here's quote from the regex' author:

> "*I do not maintain the regular expression [linked]. There may be bugs in it that have*

already been fixed in the Perl module."

.Net Address matching module

Another, shorter variant is the one used by the .Net standard library in the `EmailAddressAttribute` module:

```
^((([a-z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])+(\.([a-
z]|\d|[!#\$%&'\*\+\-\/=\?\^_`{\|}~]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])+)*)|((\x22)((((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(([\x01-\x08\x0b\x0c\x0e-
\x1f\x7f]|\x21|[\x23-\x5b]|[\x5d-\x7e]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(\\([\x01-
\x09\x0b\x0c\x0d-\x7f]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])))*(((\x20|\x09)*(\x0d\x0a))?(\x20|\x09)+)?(\x22)))@((([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])|(([a-z]|\d|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-
|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])*([a-z]|\d|[\u00A0-\uD7FF\uF900-
\uFDCF\uFDF0-\uFFEF])))\.)+(([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])|(([a-z]|[\u00A0-
\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])([a-z]|\d|-|\.|_|~|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-
\uFFEF])*([a-z]|[\u00A0-\uD7FF\uF900-\uFDCF\uFDF0-\uFFEF])))\.?$
```

But even if it's *shorter* it's still too big to be readable and easily maintainable.

Ruby Address matching module

In ruby a composition of regex are being used in the rfc822 module to match an address. This is a neat idea, as in case bugs are found, it will be easier to pinpoint the regex part to change and fix it.

Python Address matching module

As a counter example, the python email parsing module is not using a regex, but instead implements it using a parser.

Match a phone number

Here's how to match a prefix code (a `+` or (00), then a number from 1 to 1939, with an optional space):
This doesn't look for a *valid* prefix but something that might be a prefix. See the full list of prefixes

```
(?:00|\+)?[0-9]{4}
```

Then, as the entire phone number length is, at most, 15, we can look for up to 14 digits:
At least 1 digit is spent for the prefix

```
[0-9]{1,14}
```

The numbers may contains spaces, dots, or dashes and may be grouped by 2 or 3.

```
(?:[ .-][0-9]{3}){1,5}
```

With the optional prefix:

```
(?:(?:00|\+)?[0-9]{4})?(?:[ .-][0-9]{3}){1,5}
```

If you want to match a specific country format, you can use this search query and add the country, the question has certainly already been asked.

Match an IP Address

IPv4

To match IPv4 address format, you need to check for numbers `[0-9]{1,3}` three times `{3}` separated by periods `\.` and ending with another number.

```
^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$
```

This regular expression is too simple - if you want to it to be accurate, you need to check that the numbers are between `0` and `255`, with the regex above accepting `444` in any position. You want to check for 250-255 with `25[0-5]`, or any other 200 value `2[0-4][0-9]`, or any 100 value or less with `[01]?[0-9][0-9]`. You want to check that it is followed by a period `\.` three times `{3}` and then once without a period.

```
^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$
```

IPv6

IPv6 addresses take the form of 8 16-bit hex words delimited with the colon (`:`) character. In this case, we check for 7 words followed by colons, followed by one that is not. If a word has leading zeroes, they *may* be truncated, meaning each word may contain between 1 and 4 hex digits.

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$
```

This, however, is insufficient. As IPv6 addresses can become quite "wordy", the standard specifies that zero-only words may be replaced by `::`. This may only be done once in an address (for anywhere between 1 and 7 consecutive words), as it would otherwise be indeterminate. This produces a number of (rather nasty) variations:

```
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Now, putting it all together (using alternation) yields:

```
^(?:[0-9a-fA-F]{1,4}:){7}[0-9a-fA-F]{1,4}$|
^::(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}$|
```

```
^[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}$|
^[0-9a-fA-F]{1,4}:[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,3}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:){0,2}[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,4}[0-9a-fA-F]{1,4}::(?:[0-9a-fA-F]{1,4}:)?[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,5}[0-9a-fA-F]{1,4}::[0-9a-fA-F]{1,4}$|
^(?:[0-9a-fA-F]{1,4}:){0,6}[0-9a-fA-F]{1,4}::$
```

Be sure to write it out in multiline mode and with a pile of comments so whoever is inevitably tasked with figuring out what this means doesn't come after you with a blunt object.

Validate a 12hr and 24hr time string

For a 12hour time format one can use:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9]\s*[ap]m$
```

Where

- `(?:0?[0-9]|1[0-2])` is the hour
- `[-:]` is the separator, which can be adjusted to fit your need
- `[0-5][0-9]` is the minute
- `\s*[ap]m` followed any number of whitespace characters, and `am` or `pm`

If you need the seconds:

```
^(?:0?[0-9]|1[0-2])[-:][0-5][0-9][-:][0-5][0-9]\s*[ap]m$
```

For a 24hr time format:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9]$
```

Where:

- `(?:[01][0-9]|2[0-3])` is the hour
- `[-:h]` the separator, which can be adjusted to fit your need
- `[0-5][0-9]` is the minute

With the seconds:

```
^(?:[01][0-9]|2[0-3])[-:h][0-5][0-9][-:m][0-5][0-9]$
```

Where `[-:m]` is a second separator, replacing the `h` for hours with an `m` for minutes, and `[0-5][0-9]` is the second.

Match UK postcode

Regex to match postcodes in UK

The format is as follows, where A signifies a letter and 9 a digit:

Format	Coverage	Example
Cell	Cell	
AA9A 9AA	WC postcode area; EC1–EC4, NW1W, SE1P, SW1	EC1A 1BB
A9A 9AA	E1W, N1C, N1P	W1A 0AX
A9 9AA, A99 9AA	B, E, G, L, M, N, S, W	M1 1AE, B33 8TH
AA9 9AA, AA99 9AA	All other postcodes	CR2 6XH, DN55 1PT

```
(GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY])))) [0-9][A-Z-[CIKMOV]]{2})
```

Where first part:

```
 (GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[QVX]][0-9][A-HJKPSTUW])|([A-Z-[QVX]][A-Z-[IJZ]][0-9][ABEHMNPRVWXY]))))
```

Second:

```
[0-9][A-Z-[CIKMOV]]{2})
```

Read Useful Regex Showcase online: https://riptutorial.com/regex/topic/3605/useful-regex-showcase

Chapter 23: UTF-8 matchers: Letters, Marks, Punctuation etc.

Examples

Matching letters in different alphabets

Examples below are given in Ruby, but same matchers should be available in any modern language.

Let's say we have the string `"AꙊNaï ve"`, produced by Messy Artificial Intelligence. It consists of letters, but generic `\w` matcher won't match much:

```
▶ "AꙊNaï ve"[/\w+/]
#⇒ "A"
```

The correct way to match Unicode letter with combining marks is to use `\X` to specify a grapheme cluster. There is a caveat for Ruby, though. Onigmo, the regex engine for Ruby, still uses the old definition of a grapheme cluster. It is not yet updated to Extended Grapheme Cluster as defined in Unicode Standard Annex 29.

So, for Ruby we could have a workaround: `\p{L}` will do almost fine, save for it fails on combined diacritical accent on `i`:

```
▶ "AꙊNaï ve"[/\p{L}+/]
#⇒ "AꙊNai"
```

By adding the "Mark symbols" to the expression, we can finally match everything:

```
▶ "AꙊNaï ve"[/[\p{L}\p{M}]+/]
#⇒ "AꙊNaï ve"
```

Read UTF-8 matchers: Letters, Marks, Punctuation etc. online: https://riptutorial.com/regex/topic/1527/utf-8-matchers--letters--marks--punctuation-etc-

Chapter 24: When you should NOT use Regular Expressions

Remarks

Because regular expressions are limited to either a regular grammar or a context-free grammar, there are many common misuses of regular expressions. So in this topic there are a few example of when you should *NOT* use regular expressions, but use your favorite language instead.

> *Some people, when confronted with a problem, think:*
> *“I know, I'll use regular expressions.”*
> *Now they have two problems.*
> — Jamie Zawinski

Examples

Matching pairs (like parenthesis, brackets…)

Some regex engines (such as .NET) can handle context-free expressions, and will work it out. But that's not the case for most standard engines. And even if they do, you'll end up having a complex hard-to-read expression, whereas using a parsing library could make the job easier.

- How to find all possible regex matches in python?

Simple string operations

Because *Regular Expressions* can do a lot, it is tempting to use them for the simplest operations. But using a regex engine has a cost in memory and processor usage: you need to compile the expression, store the automaton in memory, initialize it and then feed it with the string to run it.

And there are many cases where it's just not necessary to use it! Whatever your language of choice is, it always has the basic string manipulation tools. So, as a rule, when there's a tool to do an action in your standard library, use that tool, not a regex:

- split a string?

For example the following snippet works in Python, Ruby and Javascript:

```
'foo.bar'.split('.')
```

Which is easier to read and understand, as well as much more efficient than the (somehow) equivalent regular expression:

```
(\w+)\.(\w+)
```

- Strip trailing spaces?

The same applies to trailing spaces!

```
'foobar     '.strip() # python or ruby
'foobar     '.trim() // javascript
```

Which would be equivalent to the following expression:

```
([^\n]*)\s*$ # keeping \1 in the substitution
```

Parsing HTML (or XML, or JSON, or C code, or…)

If you want to extract something from a webpage (or any representation/programming language), a regex is the wrong tool for the task. You should instead use your language's libraries to achieve the task.

If you want to read HTML, or XML, or JSON, just use the library that parses it properly and serves it as usable objects in your favorite language! You'll end up with readable and more maintainable code, and you won't end up

- RegEx match open tags except XHTML self-contained tags
- Python parsing HTML Using Regular Expressions
- is there a regex to generate all integers for a certain programming language

Read When you should NOT use Regular Expressions online:
https://riptutorial.com/regex/topic/4527/when-you-should-not-use-regular-expressions

Chapter 25: Word Boundary

Syntax

- POSIX style, end of word: `[[:>:]]`
- POSIX style, start of word: `[[:<:]]`
- POSIX style, word boundary: `[[:<:][:>:]]`
- SVR4/GNU, end of word: `\>`
- SVR4/GNU, start of word: `\<`
- Perl/GNU, word boundary: `\b`
- Tcl, end of word: `\M`
- Tcl, start of word: `\m`
- Tcl, word boundary: `\y`
- Portable ERE, start of word: `(^|[^[:alnum:]_])`
- Portable ERE, end of word: `([^[:alnum:]_]|$)`

Remarks

Additional Resources

- POSIX chapter on regular expressions
- Perl regular expression documentation
- Tcl re_syntax manual page
- GNU grep backslash expressions
- BSD re_format
- More reading

Examples

Match complete word

```
\bfoo\b
```

will match the complete word with no alphanumeric and _ preceding or following by it.

Taking from regularexpression.info

> There are three different positions that qualify as word boundaries:
>
> 1. Before the first character in the string, if the first character is a word character.
> 2. After the last character in the string, if the last character is a word character.
> 3. Between two characters in the string, where one is a word character and the other is not a word character.

The term *word character* here means any of the following

1. Alphabet(`[a-zA-Z]`)
2. Number(`[0-9]`)
3. Underscore _

In short, *word character* = `\w` = `[a-zA-Z0-9_]`

Find patterns at the beginning or end of a word

Examine the following strings:

```
foobarfoo
bar
foobar
barfoo
```

- the regular expression `bar` will match all four strings,
- `\bbar\b` will only match the 2nd,
- `bar\b` will be able to match the 2nd and 3rd strings, and
- `\bbar` will match the 2nd and 4th strings.

Word boundaries

The `\b` metacharacter

To make it easier to find whole words, we can use the metacharacter `\b`. It marks the **beginning** *and* the **end** of an alphanumeric sequence*. Also, since it only serves to mark this locations, it actually matches no character on its own.

**: It is common to call an alphanumeric sequence a word, since we can catch it's characters with a `\w` (the word characters class). This can be misleading, though, since `\w` also includes numbers and, in most flavors, the underscore.*

Examples:

Regex	Input	Matches?
`\bstack\b`	`stackoverflow`	**No**, since there's no ocurrence of the **whole** word `stack`
`\bstack\b`	`foo stack bar`	**Yes**, since there's nothing before nor after `stack`
`\bstack\b`	`stack!overflow`	**Yes**: there's nothing before `stack` and `!`is not a word character
`\bstack`	`stackoverflow`	**Yes**, since there's nothing before `stack`
`overflow\b`	`stackoverflow`	**Yes**, since there's nothing after `overflow`

The `\B` metacharacter

This is the opposite of `\b`, matching against the location of every non-boundary character. Like `\b`, since it matches locations, it matches no character on its own. It is useful for finding *non* whole words.

Examples:

Regex	Input	Matches?
`\Bb\B`	`abc`	**Yes**, since `b` is not surrounded by word boundaries.
`\Ba\B`	`abc`	**No**, `a` has a word boundary on its left side.
`a\B`	`abc`	**Yes**, `a` does not have a word boundary on its right side.
`\B,\B`	`a,,b`	**Yes**, it matches the second comma because `\B` will also match the space between two non-word characters (it should be noted that there is a word boundary to the left of the first comma and to the right of the second).

Make text shorter but don't break last word

To make long text at most N characters long but leave last word intact, use `.{0,N}\b` pattern:

```
^(.{0,N})\b.*
```

Read Word Boundary online: https://riptutorial.com/regex/topic/1539/word-boundary

Credits

S. No	Chapters	Contributors
1	Getting started with Regular Expressions	0rkan, Addison, balpha, Community, Confiqure, Ibrahim, J F, JelmerS, JohnLBevan, Kendra, Laurel, Maria Deleva, Mariano, Mateus, mnoronha, Rudy M, Stephen Leppik, Tot Zam, TylerH, Wolf, Yaron, zmo
2	Anchor Characters: Caret (^)	CPHPython, Eder, J F, JohnLBevan, Jojodmo, knut, Mateus, Mike H-R, Mr. Deathless, nhahtdh, revo, rgoliveira, Tom Lord, zb226
3	Anchor Characters: Dollar ($)	ArtOfCode, CPHPython, hjpotter92, Kendra, rubayet.R, Tom Lord, UNagaswamy, Wiktor Stribiżew
4	Atomic Grouping	OnlineCop
5	Back reference	Alan Moore, Kendra, OnlineCop
6	Backtracking	dorukayhan, Mike, Miljen Mikic, SQB, Thomas Ayoub, Vituel
7	Capture Groups	Addison, Alan Moore, Lucas Trzesniewski, Tomalak, Vogel612
8	Character classes	Acey, CPHPython, Dmitry Bychenko, HamZa, kdhp, Lucas Trzesniewski, Maria Deleva, RamenChef, rgoliveira, rock321987, Wiktor Stribiżew
9	Escaping	CPHPython, David Knipe, Laurel
10	Greedy and Lazy quantifiers	0rkan, Confiqure, David Knipe, GradientByte, Laurel, Mario, Mark Stewart, Nathan Arthur, nhahtdh, phatfingers, sweaver2112, Thomas Ayoub, Tim Pietzcker
11	Lookahead and Lookbehind	BoppreH, hwnd, Lucas Trzesniewski, Maria Deleva, Wiktor Stribiżew
12	Match Reset: \K	nhahtdh, Wiktor Stribiżew, Will Barnwell
13	Matching Simple Patterns	balpha, GradientByte, Graham, Joe, Mariano, rgoliveira, Tot Zam, Yaron
14	Named capture groups	Thomas Ayoub
15	Password validation regex	rock321987

16	Possessive Quantifiers	Mark Hurd, Sebastian Lenartowicz
17	Recursion	Keith Hall, Laurel, Lucas Trzesniewski, user23013
18	Regex modifiers (flags)	Eder, Mateus, Tim Pietzcker, Wiktor Stribiżew
19	Regex Pitfalls	BrightOne
20	Regular Expression Engine Types	Lucas Trzesniewski, Markus Jarderot
21	Substitutions with Regular Expressions	Mateus
22	Useful Regex Showcase	depperm, Devid Farinelli, Echelon, Herb, Kendra, Matas Vaitkevicius, nhahtdh, Sebastian Lenartowicz, Steve Chambers, Thomas Ayoub, Tomasz Jakub Rup, zmo
23	UTF-8 matchers: Letters, Marks, Punctuation etc.	mudasobwa
24	When you should NOT use Regular Expressions	dorukayhan, Kendra, zmo
25	Word Boundary	cdm, jonathanking, kdhp, Maria Deleva, Peter G, rgoliveira, Tushar