



## Examples

[Examples](#)  
[Numeric Ranges](#)  
[Floating Point Numbers](#)  
[Email Addresses](#)  
[Valid Dates](#)  
[Credit Card Numbers](#)  
[Matching Complete Lines](#)  
[Deleting Duplicate Lines](#)  
[Programming](#)  
[Two Near Words](#)

## Pitfalls

[Catastrophic Backtracking](#)  
[Making Everything Optional](#)  
[Repeated Capturing Group](#)  
[Mixing Unicode & 8-bit](#)

## More Information

[Introduction](#)  
[Quick Start](#)  
[Tutorial](#)  
[Tools and Languages](#)  
[Examples](#)  
[Books](#)  
[Reference](#)  
[Print PDF](#)  
[About This Site](#)  
[RSS Feed & Blog](#)

## Example Regexes to Match Common Programming Language Constructs

Regular expressions are very useful to [manipulate source code in a text editor](#) or in a [regex-based text processing tool](#). Most programming languages use similar constructs like keywords, comments and strings. But often there are subtle differences that make it tricky to use the correct regex. When picking a regex from the list of examples below, be sure to read the description with each regex to make sure you are picking the correct one.

Unless otherwise indicated, all examples below assume that [the dot](#) does *not* match newlines and that the [caret](#) and [dollar](#) do match at embedded line breaks. In many programming languages, this means that single line mode must be off, and multi line mode must be on.

When used by themselves, these regular expressions may not have the intended result. If a comment appears inside a string, the comment regex will consider the text inside the string as a comment. The string regex will also match strings inside comments. The solution is to use more than one regular expression, like in this pseudo-code:

```
GlobalStartPosition := 0;
while GlobalStartPosition < LengthOfText do
  GlobalMatchPosition := LengthOfText;
  MatchedRegex := NULL;
  foreach Regex in RegexList do
    Regex.StartPosition := GlobalStartPosition;
    if Regex.Match and Regex.MatchPosition < GlobalMatchPosition then
      MatchedRegex := Regex;
      GlobalMatchPosition := Regex.MatchPosition;
    endif
  endforeach
  if MatchedRegex <> NULL then
    // At this point, MatchedRegex indicates which regex matched
    // and you can do whatever processing you want depending on
    // which regex actually matched.
  endif
  GlobalStartPosition := GlobalMatchPosition;
endwhile
```

If you put a regex matching a comment and a regex matching a string in RegexList, then you can be sure that the comment regex will not match comments inside strings, and vice versa.

An alternative solution is to combine regexes: `(comment)|(string)`. The [alternation](#) has the same effect as the code snipped above. Using backreferences, you can figure out which part of the regex actually matched. The drawback of this solution is that the combined regular expression quickly becomes difficult to read or maintain.

## Comments

`#.*$` matches a single-line comment starting with a `#` and continuing until the [end of the line](#). Similarly, `//.*$` matches a single-line comment starting with `//`.

If the comment must appear at the [start of the line](#), use `^#.*$`. If only [whitespace](#) is allowed between the start of the line and the comment, use `^\s*#.*$`. Compiler directives or pragmas in C can be matched this way. Note that in this last example, any leading whitespace will be part of the regex match. Use [capturing parentheses](#) to separate the whitespace and the comment.

`/\s*.*?*/` matches a C-style multi-line comment if you turn on the option for [the dot](#) to match newlines. The general syntax is `begin.*?end`. C-style comments do not allow nesting. If the "begin" part appears inside the comment, it is ignored. As soon as the "end" part is found, the comment is closed.

If your programming language allows nested comments, there is no straightforward way to match them using a regular expression, since regular expressions cannot count. Additional logic is required.

## Strings

`"[^"\\r\n]"` matches a single-line string that does not allow the quote character to appear inside the string. Using the [negated character class](#) is more efficient than using a lazy dot. `"["]*"` allows the string to span across multiple lines.

`"[^\\"\\r\n]*(?:\\\[^\\"\\r\n]*)*"` matches a single-line string in which the quote character can appear if it is escaped by a backslash. Though this regular expression may seem more complicated than it needs to be, it is much faster than simpler solutions which can cause [a whole lot of backtracking](#) in case a double quote appears somewhere all by itself rather than part of a string. `"[^\\"\\r\n]*(?:\\\[^\\"\\r\n]*)*"` allows the string to span multiple lines.


You can adapt the above regexes to match any sequence delimited by two (possibly different) characters. If we use `b` for the starting character, `e` and the end, and `x` as the escape character, the version without escape becomes



`b[^e\r\n]*e`, and the version with escape becomes `b[^ex\r\n]*(?:x.[^ex\r\n]*)*e`.


## Numbers


---

`\b\d+\b` matches a positive integer number. Do not forget the [word boundaries](#)! `[-+]? \b\d+\b` allows for a sign.

`\b0[xX][0-9a-fA-F]+\b`  matches a C-style hexadecimal number.

`((\b[0-9]+)?\.\.?)?[0-9]+\b`  matches an integer number as well as a floating point number with optional integer part. `(\b[0-9]+\.\.(\b[0-9]+\b)?|\.\.[0-9]+\b)`  matches a floating point number with optional integer as well as optional fractional part, but does not match an integer number.

`((\b[0-9]+)?\.\.?)?\b[0-9]+([eE][-+]?[0-9]+)?\b`  matches a number in scientific notation. The mantissa can be an integer or floating point number with optional integer part. The exponent is optional.

`\b[0-9]+(\.\.[0-9]+)?(e[-+]?[0-9]+)?\b`  also matches a number in scientific notation. The difference with the previous example is that if the mantissa is a floating point number, the integer part is mandatory.

If you read through the [floating point number example](#), you will notice that the above regexes are different from what is used there. The above regexes are more [stringent](#). They use word boundaries to exclude numbers that are part of other things like identifiers. You can prepend `[-+]?` to all of the above regexes to include an optional sign in the regex. I did not do so above because in programming languages, the + and - are usually considered operators rather than signs.

## Reserved Words or Keywords

---

Matching [reserved words](#) is easy. Simply use [alternation](#) to string them together:  
`\b(first|second|third|etc)\b` Again, do not forget the [word boundaries](#).

## Make a Donation

---

Did this website just save you a trip to the bookstore? Please [make a donation](#) to support this site, and you'll get a lifetime of advertisement-free access to this site!

---

Page URL: <http://www.Regular-Expressions.info/examplesprogrammer.html>  
Page last updated: 19 June 2009  
Site last updated: 02 February 2012  
Copyright © 2003-2012 Jan Goyvaerts. All rights reserved.