

Regular expression to match a line that doesn't contain a word

Asked 10 years, 10 months ago Active 6 days ago Viewed 3.1m times

4094 ▲ I know it's possible to match a word and then reverse the matches using other tools (e.g. `grep -v`). However, is it possible to match lines that do not contain a specific word, e.g. `hede`, using a regular expression?

4094

▼
Input:

★
1546
hoho
hihi
haha
hede

Code:

```
grep "<Regex for 'doesn't contain hede'" input
```

Desired output:

hoho
hihi
haha


regex

regex-negation

edited Oct 22 at 14:00

community wiki
20 revs, 14 users 18%
knaser

78 Probably a couple years late, but what's wrong with: `([h]*(h(^e|$)|he(^d|$)|hed(^e|$))))* ?` The idea is simple. Keep matching until you see the start of the unwanted string, then only match in the N-1 cases where the string is unfinished (where N is the length of the string). These N-1 cases are "h followed by non-e", "he followed by non-d", and "hed followed by non-e". If you managed to pass these N-1 cases, you successfully *didn't* match the unwanted string so you can start looking for `[h]*` again – [stevendesu](#) Sep 29 '11 at 3:44

- 301 @stevendesu: try this for 'a-very-very-long-word' or even better half a sentence. Have fun typing. BTW, it is nearly unreadable. Don't know about the performance impact. – [Peter Schuetze](#) Jan 30 '12 at 18:45
- 13 @PeterSchuetze: Sure it's not pretty for very very long words, but it is a viable and correct solution. Although I haven't run tests on the performance, I wouldn't imagine it being too slow since most of the latter rules are ignored until you see an h (or the first letter of the word, sentence, etc.). And you could easily generate the regex string for long strings using iterative concatenation. If it works and can be generated quickly, is legibility important? That's what comments are for. – [stevendesu](#) Feb 2 '12 at 3:14
- 53 @stevendesu: i'm even later, but that answer is almost completely wrong. for one thing, it requires the subject to contain "h" which it shouldn't have to, given the task is "match lines which [do] not contain a specific word". let us assume you meant to make the inner group optional, and that the pattern is anchored: `^([^\h]*(h([^\e]|$)|he([^\d]|$)|hed([^\e]|$)))?*$` this fails when instances of "hede" are preceded by partial instances of "hede" such as in "hhede". – [jaytea](#) Sep 10 '12 at 10:41 
- 7 This question has been added to the [Stack Overflow Regular Expression FAQ](#), under "Advanced Regex-Fu". – [aliteralmind](#) Apr 10 '14 at 1:30

29 Answers

- ▲
5582
▼
✓
- The notion that regex doesn't support inverse matching is not entirely true. You can mimic this behavior by using negative look-arounds:
- ```
^((?!hede).)*$
```
- The regex above will match any string, or line without a line break, **not** containing the (sub)string 'hede'. As mentioned, this is not something regex is "good" at (or should do), but still, it *is* possible.
- And if you need to match line break chars as well, use the [DOT-ALL modifier](#) (the trailing `s` in the following pattern):
- ```
/^((?!hede).)*$/s
```
- or use it inline:
- ```
/(?s)^((?!hede).)*$/
```
- (where the `/.../` are the regex delimiters, i.e., not part of the pattern)
- If the DOT-ALL modifier is not available, you can mimic the same behavior with the character class `[\s\S]` :

```
/^((?!hede)[\s\S])*$/
```

## Explanation

A string is just a list of  $n$  characters. Before, and after each character, there's an empty string. So a list of  $n$  characters will have  $n+1$  empty strings. Consider the string "ABhedeCD" :

|     |    |   |    |   |    |   |    |   |    |   |    |   |    |   |    |   |    |
|-----|----|---|----|---|----|---|----|---|----|---|----|---|----|---|----|---|----|
| S = | e1 | A | e2 | B | e3 | h | e4 | e | e5 | d | e6 | e | e7 | C | e8 | D | e9 |
|-----|----|---|----|---|----|---|----|---|----|---|----|---|----|---|----|---|----|

|       |   |   |   |   |   |   |   |   |
|-------|---|---|---|---|---|---|---|---|
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------|---|---|---|---|---|---|---|---|

where the `e`'s are the empty strings. The regex `(?!hede).` looks ahead to see if there's no substring "hede" to be seen, and if that is the case (so something else is seen), then the `.` (dot) will match any character except a line break. Look-arounds are also called *zero-width-assertions* because they don't *consume* any characters. They only assert/validate something.

So, in my example, every empty string is first validated to see if there's no "hede" up ahead, before a character is consumed by the `.` (dot). The regex `(?!hede).` will do that only once, so it is wrapped in a group, and repeated zero or more times: `((?!hede).)*`. Finally, the start- and end-of-input are anchored to make sure the entire input is consumed: `^((?!hede).)*$`

As you can see, the input "ABhedeCD" will fail because on `e3`, the regex `(?!hede)` fails (there *is* "hede" up ahead!).

edited May 8 '17 at 20:35

community wiki  
10 revs, 6 users 83%  
Bart Kiers

- 
- 23 I would not go so far as to say that this is something regex is bad at. The convenience of this solution is pretty obvious and the performance hit compared to a programmatic search is often going to be unimportant. – [Archimaredes](#) Mar 3 '16 at 16:09
- 
- 23 Strictly speaking negative look-ahead makes you regular expression not-regular. – [Peter K](#) Nov 18 '16 at 15:03
- 
- 45 @PeterK, sure, but this is SO, not MathOverflow or CS-Stackexchange. People asking a question here are generally looking for a practical answer. Most libraries or tools (like `grep`, which the OP mentions) with regex-support all have features that mke them non-regular in a theoretical sense. – [Bart Kiers](#) Nov 18 '16 at 15:08
- 
- 17 @Bart Kiers, no offense to you answer, just this abuse of terminology irritates me a bit. The really confusing part here is that regular expressions in the strict sense can very much do what OP wants, but the common language to write them does not allow it, which leads to (mathematically ugly) workarounds like look-aheads. Please see [this answer](#) below and my comment there for (theoretically aligned) proper way of doing it. Needless to say it works faster on large inputs. – [Peter K](#) Nov 18 '16 at 15:33
-

15 In case you ever wondered how to do this in vim: `^\(\\(hede\\)\@!.\\)*$` – [Daidars](#) Nov 24 '16 at 11:58

Note that the solution to **does not start with “hede”**:

702

`^(?!hede).*`

is generally much more efficient than the solution to **does not contain “hede”**:

`^((?!hede).)*`


The former checks for “hede” only at the input string’s first position, rather than at every position.

edited Aug 27 '13 at 16:58


community wiki  
3 revs, 2 users 69%  
[FireCoding](#)

4 Thanks, I used it to validate that the string doesn't contain sequence of digits `^((?!\\d{5,}).)*` – [Samih A](#) May 10 '15 at 10:42

2 Hello! I can't compose **does not end with "hede"** regex. Can you help with it? – [Aleks Ya](#) Oct 18 '15 at 21:33

1 @AleksYa: just use the "contain" version, and include the end anchor into the search string: change the string to "not match" from "hede" to "hede\$" – [Nyerguds](#) May 4 '16 at 10:42 

1 @AleksYa: the does not end version could be done using negative lookbehind as: `(.)*(?!hede)$` . @Nyerguds' version would work as well but completely misses the point on performance the answer mentions. – [thisismydesign](#) Sep 14 '17 at 16:53

3 Why are so many answers saying `^((?!hede).)*` ? Is it not more efficient to use `^(?!.*hede).*` ? It does the same thing but in fewer steps – [JackPRead](#) Jan 15 at 10:53 

If you're just using it for grep, you can use `grep -v hede` to get all lines which do not contain hede.

197

ETA Oh, rereading the question, `grep -v` is probably what you meant by "tools options".

answered Jan 2 '09 at 7:41

community wiki  
[Athena](#)

- 20 Tip: for progressively filtering out what you don't want: `grep -v "hede" | grep -v "hihi" | ...etc.` – [Olivier Lalonde](#) May 5 '14 at 22:08
- 46 Or using only one process `grep -v -e hede -e hihi -e ...` – [Olaf Dietsche](#) Apr 26 '15 at 5:42
- 13 Or just `grep -v "hede\|hihi" :)` – [Putnik](#) Dec 9 '16 at 15:29
- 2 If you have many patterns that you want to filter out, put them in a file and use `grep -vf pattern_file file` – [codeforester](#) Mar 11 '18 at 18:35
- 2 Or simply `egrep` or `grep -Ev "hede|hihi|etc"` to avoid the awkward escaping. – [Amit Naidu](#) Jun 3 '18 at 10:54

**Answer:**

148 `^((?!hede).)*$`

**Explanation:**

`^` the beginning of the string, `(` group and capture to `\1` (0 or more times (matching the most amount possible)),  
`(?!` look ahead to see if there is not,

`hede` your string,

`)` end of look-ahead, `.` any character except `\n`,

`)*` end of `\1` (Note: because you are using a quantifier on this capture, only the LAST repetition of the captured pattern will be stored in `\1`)

`$` before an optional `\n`, and the end of the string

edited Dec 6 '17 at 11:23

community wiki  
 3 revs, 2 users 72%  
[Jessica](#)

- 13 awesome that worked for me in sublime text 2 using multiple words ' `^((?!DSAU_PW8882WEB2|DSAU_PW8884WEB2|DSAU_PW8884WEB).)*$` ' – [Damodar Bashyal](#) Aug 11 '15 at 2:07
- 2 @DamodarBashyal I know I'm pretty late here, but you could totally remove the second term there and you would get the exact same results – [forresthopkinsa](#) Jun 12 '17 at 16:19



The given answers are perfectly fine, just an academic point:

97

Regular Expressions in the meaning of theoretical computer sciences *ARE NOT ABLE* do it like this. For them it had to look something like this:



```
^([\^h].*$)|(\h([\^e].*$|$))|(\he([\^h].*$|$))|(\heh([\^e].*$|$))|(\hehe.+$)
```

This only does a FULL match. Doing it for sub-matches would even be more awkward.

answered Sep 2 '11 at 15:53

community wiki  
Hades32

- 
- 1 Important to note this only uses basic POSIX.2 regular expressions and thus whilst terse is more portable for when PCRE is not available. – [Steve-o](#) Feb 19 '14 at 17:25
- 
- 5 I agree. Many if not most regular expressions are not regular languages and could not be recognized by a finite automata. – [ThomasMcLeod](#) Mar 22 '14 at 21:36
- 
- @ThomasMcLeod, Hades32: Is it within the realms of any possible regular language to be able to say '**not**' and '**and**' as well as the '**or**' of an expression such as ' (hede|Hihi) '? (*This maybe a question for CS.*) – [James Haigh](#) Jun 13 '14 at 16:54
- 
- 7 @JohnAllen: **ME!!!** ...Well, not the actual regex but the academic reference, which also relates closely to computational complexity; PCREs fundamentally can not guarantee the same efficiency as POSIX regular expressions. – [James Haigh](#) Jun 13 '14 at 17:04
- 
- 4 Sorry -this answer just doesn't work, it will match hhehe and even match hehe partially (the second half) – [Falco](#) Aug 13 '14 at 12:57
- 



If you want the regex test to **only** fail if the *entire string* matches, the following will work:

55

```
^(?!hede$).*
```



e.g. -- If you want to allow all values except "foo" (i.e. "foofoo", "barfoo", and "foobar" will pass, but "foo" will fail), use: `^(?!foo$).*`

Of course, if you're checking for *exact* equality, a better general solution in this case is to check for string equality, i.e.

```
myStr !== 'foo'
```

You could even put the negation *outside* the test if you need any regex features (here, case insensitivity and range matching):

```
!/^[a-f]oo$/i.test(myStr)
```

The regex solution at the top of this answer may be helpful, however, in situations where a positive regex test is required (perhaps by an API).

edited Nov 7 '18 at 21:51

community wiki

9 revs

Roy Tinker

---

what about trailing whitespaces? Eg, if I want test to fail with string " hede " ? – [eagor](#) May 12 '17 at 9:45

---

@eagor the `\s` directive matches a single whitespace character – [Roy Tinker](#) May 12 '17 at 21:07

---

thanks, but I didn't manage to update the regex to make this work. – [eagor](#) May 13 '17 at 19:22

---

2 @eagor: `^(?!\\s*hede\\s*$).*` – [Roy Tinker](#) May 15 '17 at 17:33

---



53



FWIW, since regular languages (aka rational languages) are closed under complementation, it's always possible to find a regular expression (aka rational expression) that negates another expression. But not many tools implement this.

[Vcsn](#) supports this operator (which it denotes `{c}` , postfix).

You first define the type of your expressions: labels are letter ( `lal_char` ) to pick from `a` to `z` for instance (defining the alphabet when working with complementation is, of course, very important), and the "value" computed for each word is just a Boolean: `true` the word is accepted, `false` , rejected.

In Python:

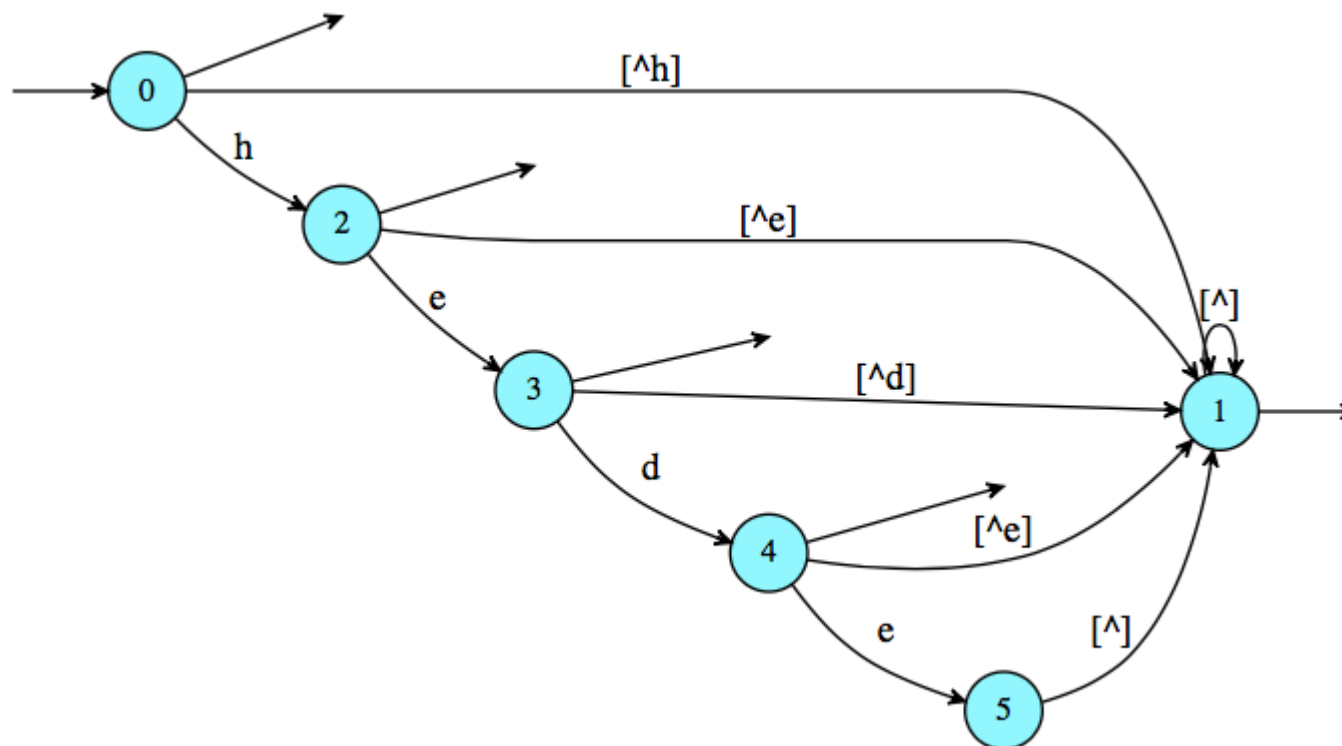
```
In [5]: import vcsn
 c = vcsn.context('lal_char(a-z), b')
 c
Out[5]: {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z} → \mathbb{B}
```

then you enter your expression:

```
In [6]: e = c.expression('(hede){c}'); e
Out[6]: (hede)^c
```

convert this expression to an automaton:

```
In [7]: a = e.automaton(); a
```



finally, convert this automaton back to a simple expression.

```
In [8]: print(a.expression())
\eh(\e+e(\e+d))+([^h]+h([^e]+e([^d]+d([^e]+e[^])))][^]*
```

where `+` is usually denoted `|`, `\e` denotes the empty word, and `[^]` is usually written `.` (any character). So, with a bit of rewriting `()|h(ed)?|([^[h]|h([^[e]|e([^[d]|d([^[e]|e.))]))][^]*`.

You can see this example [here](#), and try Vcsn online [there](#).

edited Jan 8 '18 at 23:58

community wiki  
5 revs, 2 users 98%  
akim



6 True, but ugly, and only doable for small character sets. You don't want to do this with Unicode strings :-)) – [reinierpost](#) Nov 8 '15 at 23:43

There are more tools that allow it, one of the most impressive being [Ragel](#). There it would be written as `(any* - ('hehe' any*))` for start-aligned match or `(any* -- ('hehe' any*))` for unaligned. – [Peter K](#) Nov 18 '16 at 15:09

1 @reinierpost: why is it ugly and what's the problem with unicode? I can't agree on both. (I have no experience with vcsn, but have with DFA). – [Peter K](#) Nov 18 '16 at 15:39

The regexp `()|h(ed?)?|([^\h]|h([^\e]|e([^\d]|d([^\e]|e.))))).*` didn't work for me using `egrep`. It matches `hede`. I also tried anchoring it to the beginning and end, and it still didn't work. – [Pedro Gimeno](#) Dec 6 '16 at 23:18

3 @PedroGimeno When you anchored, you made sure to put this regex in parens first? Otherwise the precedences between anchors and `|` won't play nicely. `'^^(())|h(ed?)?|([^\h]|h([^\e]|e([^\d]|d([^\e]|e.))))).*'$`. – [akim](#) Dec 8 '16 at 9:03

Here's [a good explanation](#) of why it's not easy to negate an arbitrary regex. I have to agree with the other answers, though: if this is anything other than a hypothetical question, then a regex is not the right choice here.

answered Jan 2 '09 at 8:03

community wiki  
[Josh Lee](#)

10 Some tools, and specifically mysqldumpslow, only offer this way to filter data, so in such a case, finding a regex to do this is the best solution apart from rewriting the tool (various patches for this have not been included by MySQL AB / Sun / Oracle. – [FGM](#) Aug 7 '12 at 12:21

1 Exactly analagous to my situation. Velocity template engine uses regular expressions to decide when to apply a transformation (escape html) and I want it to always work EXCEPT in one situation. – [Henno Vermeulen](#) Oct 18 '13 at 14:43

1 What alternative is there? Ive never encountered anything that could do precise string matching besides regex. If OP is using a programming language, there may be other tools available, but if he/she is using not writing code, there probably isnt any other choice. – [kingfrito\\_5005](#) Oct 20 '16 at 18:32

1 One of many non-hypothetical scenarios where a regex is the best available choice: I'm in an IDE (Android Studio) that shows log output, and the only filtering tools provided are: plain strings, and regex. Trying to do this with plain strings would be a complete fail. – [LarsH](#) Dec 5 '16 at 16:11

With negative lookahead, regular expression can match something not contains specific pattern. This is answered and explained by Bart Kiers. Great explanation!

However, with Bart Kiers' answer, the lookahead part will test 1 to 4 characters ahead while matching any single character. We can avoid this and let the lookahead part check out the whole text, ensure there is no 'hede', and then the normal part `(.*)` can eat the whole text all at one time.

Here is the improved regex:

```
/^(?!.*?hede).*$
```

Note the `(*)` lazy quantifier in the negative lookahead part is optional, you can use `(*)` greedy quantifier instead, depending on your data: if 'hede' does present and in the beginning half of the text, the lazy quantifier can be faster; otherwise, the greedy quantifier be faster. However if 'hede' does not present, both would be equal slow.

Here is the [demo code](#).

For more information about lookahead, please check out the great article: [Mastering Lookahead and Lookbehind](#).

Also, please check out [RegexGen.js](#), a JavaScript Regular Expression Generator that helps to construct complex regular expressions. With RegexGen.js, you can construct the regex in a more readable way:

```
var _ = regexGen;

var regex = _(
 _.startOfLine(),
 _.anything().notContains(// match anything that not contains:
 _.anything().lazy(), 'hede' // zero or more chars that followed by 'hede',
 // i.e., anything contains 'hede'
),
 _.endOfLine()
);
```

answered Jul 14 '14 at 18:21

community wiki  
[amobiz](#)

3 so to simply check if given string does not contain str1 and str2: `^(?!.*(str1|str2)).*$` – [S.Serpooshan](#) Mar 1 '17 at 7:20

1 Yes, or you can use lazy quantifier: `^(?!.*(?:str1|str2)).*$`, depending on your data. Added the `?:` since we don't need to capture it. – [amobiz](#) Mar 2 '17 at 9:59

This is by far the best answer by a factor of 10xms. If you added your jsfiddle code and results onto the answer people might notice it. I wonder why the lazy version is faster than the greedy version when there is no hede. Shouldn't they take the same amount of time? – [user5389726598465](#) Jul 23 '17 at 9:06

Yes, they take the same amount of time since they both tests the whole text. – [amobiz](#) Aug 3 '17 at 3:50

## Benchmarks

41

I decided to evaluate some of the presented Options and compare their performance, as well as use some new Features. Benchmarking on .NET Regex Engine: <http://regexhero.net/tester/>

### Benchmark Text:

The first 7 lines should not match, since they contain the searched Expression, while the lower 7 lines should match!

[illegible]

```
Regex Her
egex Hero
egex Hero is a real-time online Silverlight Regular Expression Tester.
Regex Her is a real-time online Silverlight Regular Expression Tester.
Regex Her Regex Her Regex Her Regex Her Regex Her Regex Her is a real-time online
Silverlight Regular Expression Tester.
Nobody is a real-time online Silverlight Regular Expression Tester.
Regex Her o egex Hero Regex Hero Reg ex Hero is a real-time online Silverlight Regular
Expression Tester.
```

## Results:

Results are Iterations per second as the median of 3 runs - **Bigger Number = Better**

|     |                                            |       |                                                          |
|-----|--------------------------------------------|-------|----------------------------------------------------------|
| 01: | <code>^(?!Regex Hero).*</code>             | 3.914 | // Accepted Answer                                       |
| 02: | <code>^(?:(!Regex Hero).*</code>           | 5.034 | // With Non-Capturing group                              |
| 03: | <code>^(?&gt;[R]+ R(!egex Hero)).*</code>  | 6.137 | // Lookahead only on the right first Letter              |
| 04: | <code>^(?&gt;(?:.*?Regex Hero)?)^.*</code> | 7.426 | // Match the word and check if you're still at linestart |

```
05: ^(? (?!.*?Regex Hero)(?#fail)|.*)$ 7.371 // Logic Branch: Find Regex Hero?
match nothing, else anything
```

```
P1: ^(? (?!.*?Regex Hero)(*FAIL)|(*ACCEPT)) ???? // Logic Branch in Perl - Quick FAIL
P2: .*?Regex Hero(*COMMIT)(*FAIL)|(*ACCEPT) ???? // Direct COMMIT & FAIL in Perl
```

Since .NET doesn't support action Verbs (\*FAIL, etc.) I couldn't test the solutions P1 and P2.

## Summary:


I tried to test most proposed solutions, some Optimizations are possible for certain words. For Example if the First two letters of the search string are not the Same, answer 03 can be expanded to `^(?>[^R]+|R+(?!egex Hero))*$` resulting in a small performance gain.

But the overall most readable and performance-wise fastest solution seems to be 05 using a conditional statement or 04 with the possessive quantifier. I think the Perl solutions should be even faster and more easily readable.

answered Aug 13 '14 at 14:58

community wiki  
Falco

---

4 You should time `^(?!.*hede)` too. /// Also, it's probably better to rank the expressions for the matching corpus and the non-matching corpus separately because it's usually a case that most line match or most lines don't. – [ikegami](#) Aug 23 '16 at 0:07 

---



Not regex, but I've found it logical and useful to use serial greps with pipe to eliminate noise.

32

eg. search an apache config file without all the comments-



```
grep -v '\#' /opt/lampp/etc/httpd.conf # this gives all the non-comment lines
```

and

```
grep -v '\#' /opt/lampp/etc/httpd.conf | grep -i dir
```

The logic of serial grep's is (not a comment) and (matches dir)

edited Mar 17 '11 at 20:19

community wiki

2 I think he is asking for the regex version of the `grep -v` – [Angel.King.47](#) Jul 12 '11 at 15:27

9 This is dangerous. Also misses lines like `good_stuff #comment_stuff` – [Xavi Montero](#) Mar 1 '13 at 19:54

with this, you avoid to test a lookahead on each positions:

29 `/^(?:[^\h]+|h+(?!ede))*$/`

equivalent to (for .net):

`^(?>(?:[^\h]+|h+(?!ede)))*$`

Old answer:

`/^(?>[^\h]+|h+(?!ede))*$/`

edited Jun 4 '18 at 10:00

community wiki  
11 revs, 2 users 96%  
[Casimir et Hippolyte](#)

7 Good point; I'm surprised nobody mentioned this approach before. However, that particular regex is prone to [catastrophic backtracking](#) when applied to text that doesn't match. Here's how I would do it: `/^[^\h]*(?:h+(?!ede)[^\h]*)*$/` – [Alan Moore](#) Apr 14 '13 at 5:26

...or you can just make all the quantifiers possessive. ;) – [Alan Moore](#) Apr 15 '13 at 15:17 

@Alan Moore - I'm surprised too. I saw your comment (and best regex in the pile) here only after posting this same pattern in an answer below. – [ridgerunner](#) Dec 20 '13 at 3:08

@ridgerunner, doesn't have to be the best tho. I've seen benchmarks where the top answer performs better. (I was surprised about that tho.) – [Qtax](#) Feb 20 '14 at 13:10

Aforementioned `(?: (? ! hede) . ) *` is great because it can be anchored.

20

```
^(?:(!hede).)*$ # A Line without hede
foo(?:(!hede).)*bar # foo followed by bar, without hede between them
```

But the following would suffice in this case:

```
^(?!.*hede) # A Line without hede
```

This simplification is ready to have "AND" clauses added:

```
^(?!.*hede)(?=.*foo)(?=.*bar) # A Line with foo and bar, but without hede
^(?!.*hede)(?=.*foo).*bar # Same
```

edited Aug 23 '16 at 0:10

community wiki  
2 revs  
ikegami

Here's how I'd do it:

19

```
^[^h]*(h(!ede)[^h]*)*$
```

Accurate and more efficient than the other answers. It implements Friedl's *"unrolling-the-loop"* efficiency technique and requires much less backtracking.

answered Dec 20 '13 at 3:03

community wiki  
ridgerunner

If you want to match a character to negate a word similar to negate character class:

17

For example, a string:

```
<?
$str="aaa bbb4 aaa bbb7";
```

```
?>
```

Do not use:

```
<?
preg_match('/aaa[^bbb]+?bbb7/s', $str, $matches);
?>
```

Use:

```
<?
preg_match('/aaa(?:?!bbb).)+?bbb7/s', $str, $matches);
?>
```

Notice "(?!bbb)." is neither lookbehind nor lookahead, it's lookcurrent, for example:

```
"(?:=abc)abcde", "(?!abc)abcde"
```

edited Apr 3 '14 at 16:17

community wiki  
5 revs, 3 users 86%  
diyism

- 
- 3 There is no "lookcurrent" in perl regexp's. This is truly a negative lookahead (prefix (?!)). Positive lookahead's prefix would be (?= while the corresponding lookbehind prefixes would be (?<! and (?<= respectively. A lookahead means that you read the next characters (hence "ahead") without consuming them. A lookbehind means that you check characters that have already been consumed. – [Didier L](#) May 21 '12 at 16:35
- 



An, in my opinion, more readable variant of the top answer:

14

```
^(?!.*hede)
```



Basically, "match at the beginning of the line if and only if it does not have 'hede' in it" - so the requirement translated almost directly into regex.

Of course, it's possible to have multiple failure requirements:

```
^(?!.*(hede|hodo|hada))
```

**Details:** The ^ anchor ensures the regex engine doesn't retry the match at every location in the string, which would match every string.

The ^ anchor in the beginning is meant to represent the beginning of the line. The grep tool matches each line one at a time, in contexts where you're working with a multiline string, you can use the "m" flag:

```
/^(?!.*hede)/m # JavaScript syntax
```

or

```
(?m)^(?!.*hede) # Inline flag
```

edited Dec 8 '18 at 20:18

community wiki  
5 revs, 2 users 54%  
Dannie P

---

Excellent example with multiple negation. – [Peter Parada](#) Jul 11 at 16:50

---

One difference from top answer is that this does not match anything, and that matches the whole line if without "hede" – [Z. Khullah](#) Aug 20 at 19:33

---

▲ The OP did not specify or Tag the post to indicate the context (programming language, editor, tool) the Regex will be used within.

13

▼ For me, I sometimes need to do this while editing a file using `Textpad`.

`Textpad` supports some Regex, but does not support lookahead or lookbehind, so it takes a few steps.

If I am looking to retain all lines that **Do NOT** contain the string `hede`, I would do it like this:

1. Search/replace the entire file to add a unique "Tag" to the beginning of each line containing any text.

```
Search string:^(.)
Replace string:<@#-unique-#@>\1
Replace-all
```



2. Delete all lines that contain the string `hede` (replacement string is empty):

```
Search string:<@#-unique-#@>.*hede.*\n
Replace string:<nothing>
Replace-all
```

3. At this point, all remaining lines **Do NOT** contain the string `hede`. Remove the unique "Tag" from all lines (replacement string is empty):

```
Search string:<@#-unique-#@>
Replace string:<nothing>
Replace-all
```

Now you have the original text with all lines containing the string `hede` removed.

If I am looking to **Do Something Else** to only lines that **Do NOT** contain the string `hede`, I would do it like this:

1. Search/replace the entire file to add a unique "Tag" to the beginning of each line containing any text.

```
Search string:^(.)
Replace string:<@#-unique-#@>\1
Replace-all
```

2. For all lines that contain the string `hede`, remove the unique "Tag":

```
Search string:<@#-unique-#@>(.*hede)
Replace string:\1
Replace-all
```

3. At this point, all lines that begin with the unique "Tag", **Do NOT** contain the string `hede`. I can now do my **Something Else** to only those lines.

4. When I am done, I remove the unique "Tag" from all lines (replacement string is empty):

```
Search string:<@#-unique-#@>
Replace string:<nothing>
Replace-all
```

edited Apr 26 '13 at 22:46

community wiki  
2 revs  
Kevin Fegan



10



Since the introduction of ruby-2.4.1, we can use the new [Absent Operator](#) in Ruby's Regular Expressions from the official [doc](#)

```
(?~abc) matches: "", "ab", "aab", "cccc", etc.
It doesn't match: "abc", "aabc", "ccccabc", etc.
```

Thus, in your case `^(?~hede)$` does the job for you

```
2.4.1 :016 > ["hoho", "hihi", "haha", "hede"].select{|s| /^(?~hede)$/.match(s)}
=> ["hoho", "hihi", "haha"]
```

answered Mar 23 '17 at 13:42

community wiki  
aelor



9



Through PCRE verb `(*SKIP)(*F)`

```
^hede$(*SKIP)(*F)|^.*$
```

This would completely skips the line which contains the exact string `hede` and matches all the remaining lines.

[DEMO](#)

## Execution of the parts:

Let us consider the above regex by splitting it into two parts.

1. Part before the `|` symbol. Part **shouldn't be matched**.

```
^hede$(*SKIP)(*F)
```

2. Part after the `|` symbol. Part **should be matched**.

```
^.*$
```

## PART 1

Regex engine will start its execution from the first part.

```
^hede$(*SKIP)(*F)
```

### Explanation:

- `^` Asserts that we are at the start.
- `hede` Matches the string `hede`
- `$` Asserts that we are at the line end.

So the line which contains the string `hede` would be matched. Once the regex engine sees the following `(*SKIP)(*F)` (*Note: You could write `(*F)` as `(*FAIL)`*) verb, it skips and make the match to fail. `|` called alteration or logical OR operator added next to the PCRE verb which inturn matches all the boundaries exists between each and every character on all the lines except the line contains the exact string `hede` . See the demo [here](#). That is, it tries to match the characters from the remaining string. Now the regex in the second part would be executed.

## PART 2

```
^.*$
```

### Explanation:

- `^` Asserts that we are at the start. ie, it matches all the line starts except the one in the `hede` line. See the demo [here](#).

- `.*` In the Multiline mode, `.` would match any character except newline or carriage return characters. And `*` would repeat the previous character zero or more times. So `.*` would match the whole line. See the demo [here](#).

## Hey why you added .\* instead of .+ ?

Because `.*` would match a blank line but `.+` won't match a blank. We want to match all the lines except `hede`, there may be a possibility of blank lines also in the input. so you must use `.*` instead of `.+`. `.+` would repeat the previous character one or more times. See `.*` matches a blank line [here](#).

- \$ End of the line anchor is not necessary here.

edited Oct 9 '14 at 7:51

community wiki  
2 revs  
Avinash Raj

Since no one else has given a direct answer to the question *that was asked*, I'll do it.

9

The answer is that with POSIX `grep`, it's impossible to literally satisfy this request:

```
grep "<Regex for 'doesn't contain hede'" input
```

The reason is that POSIX `grep` is only required to work with [Basic Regular Expressions](#), which are simply not powerful enough for accomplishing that task (they are not capable of parsing regular languages, because of lack of alternation and parentheses).

However, GNU `grep` implements extensions that allow it. In particular, `\|` is the alternation operator in GNU's implementation of BREs, and `\(` and `\)` are the parentheses. If your regular expression engine supports alternation, negative bracket expressions, parentheses and the Kleene star, and is able to anchor to the beginning and end of the string, that's all you need for this approach. Note however that negative sets `[^ ... ]` are very convenient in addition to those, because otherwise, you need to replace them with an expression of the form `(a|b|c| ... )` that lists every character that is not in the set, which is extremely tedious and overly long, even more so if the whole character set is Unicode.

With GNU `grep`, the answer would be something like:

```
grep "\^[^h]\|h(h\|eh\|edh)*\^[^eh]\|e[^dh]\|ed[^eh]\|\)\)*\(\|h(h\|eh\|edh)*\|e\|ed\)\)$" input
```

(found with [Grail](#) and some further optimizations made by hand).

You can also use a tool that implements [Extended Regular Expressions](#), like `egrep`, to get rid of the backslashes:

```
egrep "^[^h]|h(h|eh|edh)*([^eh]|e[^dh]|ed[^eh])*(|h(h|eh|edh)*(|e|ed))$" input
```

Here's a script to test it (note it generates a file `testinput.txt` in the current directory):

```
#!/bin/bash
REGEX="^[^h]|h(h|eh|edh)*([^eh]|e[^dh]|ed[^eh])*(|h(h|eh|edh)*(|e|ed))$"

First four lines as in OP's testcase.
cat > testinput.txt <<EOF
hoho
hihi
haha
hede

h
he
ah
head
ahead
ahed
aheda
ahede
hhede
hehede
hedhede
hehehehehehedehehe
hedecidedthat
EOF
diff -s -u <(grep -v hede testinput.txt) <(grep "$REGEX" testinput.txt)
```

In my system it prints:

```
Files /dev/fd/63 and /dev/fd/62 are identical
```

as expected.

For those interested in the details, the technique employed is to convert the regular expression that matches the word into a finite automaton, then invert the automaton by changing every acceptance state to non-acceptance and vice versa, and then converting the resulting FA back to a regular expression.

Finally, as everyone has noted, if your regular expression engine supports negative lookahead, that simplifies the task a lot. For example, with GNU grep:

```
grep -P '^(?!hede).*' input
```

**Update:** I have recently found Kendall Hopkins' excellent [FormalTheory](http://www.formauri.es/personal/pgimeno/misc/non-match-regex/) library, written in PHP, which provides a functionality similar to Grail. Using it, and a simplifier written by myself, I've been able to write an online generator of negative regular expressions given an input phrase (only alphanumeric and space characters currently supported): <http://www.formauri.es/personal/pgimeno/misc/non-match-regex/>

For hede it outputs:

```
^([\^h]|h(h|e(h|dh))*([\^eh]|e([\^dh]|d[\^eh])))*(h(h|e(h|dh))*(ed?))?.$
```

which is equivalent to the above.

edited Oct 22 at 14:07

community wiki

5 revs

Pedro Gimeno



It may be more maintainable to two regexes in your code, one to do the first match, and then if it matches run the second regex to check for outlier cases you wish to block for example `^.*(hede).*` then have appropriate logic in your code.

7



OK, I admit this is not really an answer to the posted question posted and it may also use slightly more processing than a single regex. But for developers who came here looking for a fast emergency fix for an outlier case then this solution should not be overlooked.

edited Sep 13 '16 at 13:55

community wiki

2 revs, 2 users 80%

andrew pate



The [TXR Language](#) supports regex negation.

5



```
$ txr -c '@(repeat)
@{nothede /~hede/}
```

```
@(do (put-line nothede))
@(end)' Input
```

A more complicated example: match all lines that start with `a` and end with `z`, but do not contain the substring `hede`:

```
$ txr -c '@(repeat)
@{nothede /a.*z&~.*hede.*}/}
@(do (put-line nothede))
@(end)' -
az <- echoed
az
abcz <- echoed
abcz
abhederz <- not echoed; contains hede
ahedez <- not echoed; contains hede
ace <- not echoed; does not end in z
ahedz <- echoed
ahedz
```

Regex negation is not particularly useful on its own but when you also have intersection, things get interesting, since you have a full set of boolean set operations: you can express "the set which matches this, except for things which match that".

answered Jun 25 '14 at 1:23

community wiki  
Kaz

---

Note that it is also the solution for Elasticsearch Lucene based regex. – Wiktor Stribizew Feb 19 '18 at 7:30

---

The below function will help you get your desired output

4

```
<?PHP
function removePrepositions($text){
 $propositions=array('/\bfor\b/i','/\bthe\b/i');

 if(count($propositions) > 0) {
 foreach($propositions as $exceptionPhrase) {
 $text = preg_replace($exceptionPhrase, '', trim($text));
 }
 }
}
```

```

 $retval = trim($text);
}
return $retval;
}

```

```
?>
```

edited Mar 11 '17 at 1:42

community wiki  
2 revs, 2 users 91%  
Daniel Nyamasyo



2

^((?!hede).)\*\$ is an elegant solution, except since it consumes characters you won't be able to combine it with other criteria. For instance, say you wanted to check for the non-presence of "hede" and the presence of "haha." This solution would work because it won't consume characters:



^(?!.\bhede\b)(?=.\*\bhaha\b)

answered Mar 26 at 12:21

community wiki  
cloudhopperpilot



1

Another option is that to add a positive look-ahead and check if `hehe` is anywhere in the input line, then we would negate that, with an expression similar to:



^(?! (?=.\*\bhede\b)).\*\$

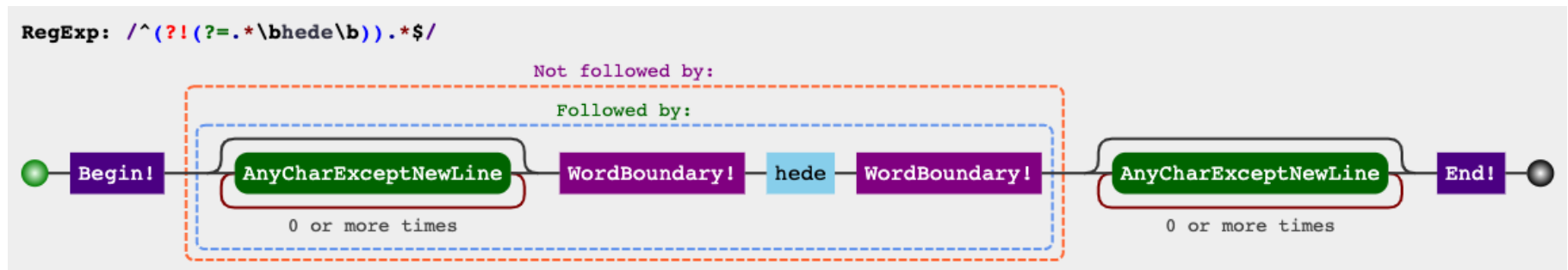
with word boundaries.

The expression is explained on the top right panel of [regex101.com](https://regex101.com), if you wish to explore/simplify/modify it, and in [this link](#), you can watch how it would match against some sample inputs, if you like.

## RegEx Circuit



[jex.im](http://jex.im) visualizes regular expressions:



answered Aug 1 at 2:36

community wiki  
Emma

## How to use PCRE's backtracking control verbs to match a line not containing a word

1 Here's a method that I haven't seen used before:

```
/. *hede(*COMMIT)^|/
```

### How it works

First, it tries to find "hede" somewhere in the line. If successful, at this point, `(*COMMIT)` tells the engine to, not only not backtrack in the event of a failure, but also not to attempt any further matching in that case. Then, we try to match something that cannot possibly match (in this case, `^`).

If a line does not contain "hede" then the second alternative, an empty subpattern, successfully matches the subject string.

This method is no more efficient than a negative lookahead, but I figured I'd just throw it on here in case someone finds it nifty and finds a use for it for other, more interesting applications.

answered Oct 11 '17 at 10:12

community wiki  
jaytea



Maybe you'll find this on Google while trying to write a regex that is able to match segments of a line (as opposed to entire lines) which do **not** contain a substring. Took me a while to figure out, so I'll share:

1



Given a string:

```
barfoobaz
```

I want to match `<span>` tags which do not contain the substring "bad".

`/<span(?:?!bad).)*?>` will match `<span class="good">` and `<span class="ugly">`.

Notice that there are two sets (layers) of parentheses:

- The innermost one is for the negative lookahead (it is not a capture group)
- The outermost was interpreted by Ruby as capture group but we don't want it to be a capture group, so I added `?:` at it's beginning and it is no longer interpreted as a capture group.

Demo in Ruby:

```
s = 'barfoobaz'
s.scan(/<span(?:?!bad).)*?/)
=> ["", ""]
```

answered Apr 25 '18 at 18:15

community wiki  
BrunoFacca



0



With [ConyEdit](#), you can use the command line `cc.gl !/hede/` to get lines that do not contain the regex matching, or use the command line `cc.dl /hede/` to delete lines that contain the regex matching. They have the same result.

answered Jul 9 '18 at 17:08

community wiki  
Donald



A simpler solution is to use the not operator !

0

Your **if** statement will need to match "contains" and not match "excludes".

```
var contains = /abc/;
var excludes = /hede/;

if(string.match(contains) && !(string.match(excludes))){ //proceed...
```

I believe the designers of RegEx anticipated the use of not operators.

edited Sep 13 '16 at 14:06

community wiki

[2 revs](#)

[user1691651-John](#)

**protected** by [Community ♦](#) Oct 8 '11 at 12:34

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 [reputation](#) on this site (the [association bonus does not count](#)).

Would you like to answer one of these [unanswered questions](#) instead?