The Greatest Regex Trick Ever

So you're doubtful at the mention of a "best regex trick"?

Fine. I'll concede right away that deciding what constitutes the best technique in any field is a curly matter. When you start out with regex, learning that the lazy question mark in `<tag>.*?</tag>` prevents you from steamrolling from the start to the end of a string such as *<tag>Tarzan</tag> likes <tag>Jane</tag>* may seem like the best regex trick ever. At other points in your career, you'll surely fall in love with regex bits such as `[^"]+` to match all the content between certain delimiters (in this case double quotes), or with atomic groups.

However, as you mature as a regex practitioner, you come to regard these techniques for what they are: language features rather than tricks. They are neat, to be sure, but they are how regex works, and nothing more.

In contrast, a "trick" is not a single point of syntax such as a negated character class or a lazy quantifier. A regex trick uses regex grammar to compose a "phrase" that achieves certain goals.

With regex there's always more to learn, and there's always a more clever person than you (unless you're the lone guy sitting on top of the mountain), so I've often been exposed to awesome tricks that were out of my league—for instance the famous regex to validate that a number is prime, or some fiendish uses of recursion. But however clever these tricks, I would not call any of them the "best regex trick ever", for the simple reason that they are one-off techniques with limited scope. You are unlikely to ever use them.

In contrast, the reason I drum up the technique on this page as the "best regex trick ever" is that it has several properties:

✽ Anyone can learn it. You don't have to be a regex master.
✽ It answers not *one*, but *several* common and practical regex questions.
✽ These questions are ones that even competent regex coders often have trouble answering gracefully.
✽ It is simple to implement in most programming languages.
✽ It is easy to extend when requirements change.
✽ It is portable over numerous regex flavors.
✽ It is usually more efficient than competing methods.
✽ It is too little-known. At least, until now.

Do I have your attention yet?

Before we proceed, I should point out some limitations of the technique:

✽ It will not butter the reverse side of a toast.
✽ It will not make small talk with your mother-in-law.
✽ It relies on your ability to inspect Group 1 captures (at least in the generic flavor), so it will not work in a non-programming environment, such as a text editor's search-and-replace function or a *grep* command.
✽ The point above also means that you may have to write one or two extra lines of code, but that is a light price to pay for a much cleaner, lighter and easier to maintain regex. Code samples for the six typical situations are provided below.
✽ There is an edge case to keep in mind. The regex engine dumps unwanted content into a trash can. In a typical context that is no problem, but if you are working with an enormous file, the trash can may get so large that you could run into memory issues.

Other than that, it's awesome. Okay, let's dive in. No need to buckle up, the technique itself is delightfully simple.

Excluding certain Contexts while Matching or Replacing

Here are some of the questions that our regex trick is able to answer with speed and grace:

✽ **How do I match a word unless it's surrounded by quotes?**
✽ **How do I match *xyz* except in contexts a, b or c?**
✽ **How do I match every word except those on a blacklist (or other contexts)?**
✽ **How do I ignore all content that is bolded (… and other contexts)?**

Once you grasp the technique, you will see that under a certain light, these are all nearly the same question.

For convenience, here are some jumping points. For full potency, I recommend you read the whole article in sequence. But if you don't care about the typical solutions to the problems addressed by the technique, you can skip directly to the description of the trick.

This is a long page. It's sure to have typos and perhaps bugs. Will you do me a favor and report any typos or bugs you find? Thanks!

The Typical Solutions

To see how convenient the trick is, it helps to first see how inconvenient some matching tasks can be when you don't know it. So let's see what other solutions exist. We'll look at two broad cases:

A. Simple Case: fixed-width non-match, as in *"Tarzan"*
First, let's examine a "simple case": we want to match *Tarzan* except when this exact word is in double-quotes. In other words, we want to exclude *"Tarzan"*.

Option 1: Lookarounds
At first you may think of framing *Tarzan* between a negative lookbehind and a negative lookahead:

```
(?<!")Tarzan(?!")
```

However, this does not work because it also excludes valid strings such as *"Tarzan and Jane"* and *"Jane and Tarzan"*, whereas we only wanted to exclude *"Tarzan"*.

Back to the Future Regex I
To account for this, you might inject a lookahead inside your negative lookbehind. This is what I call a *"Back to the Future"* regex. The lookahead inside the lookbehind asserts that after we've found the opening double quote behind *Tarzan*, we can find *Tarzan* (surprise) and a closing double quote. Since we're inside a negative lookbehind, this whole package is what we don't want.

```
(?<!"(?=Tarzan"))Tarzan
```

Step Forward then Backflip
This approach is closely related to the *Back to the Future* approach. You match *Tarzan*, then you exclude the match if it is followed by a double quote (lookahead) that is preceded by the string *"Tarzan"*.

```
Tarzan(?!"(?<="Tarzan"))
```

Conditional
Alternately, you might first turn the negative lookbehind into a positive lookbehind that captures the opening quote if found, then tag a conditional at the end to assert that if Group 1 was set, the following character cannot be a double quote.

```
(?>(?<=(")|))Tarzan(?(1)(?!"))
```

Logic à la Lewis Carroll
For this simple sample problem, you can modifiy the faulty *lookarounds* solution with a bit of logic:

```
(?<!")Tarzan|Tarzan(?!")
```

The left side of the alternation excludes *"Tarzan*, but the right side allows it. The right side of the alternation excludes *Tarzan"*, but the left side allows it. As desired, this expression can match *Tarzan*, *"Tarzan*, *Tarzan"* but not *"Tarzan"*. This is neat, but is it obvious? You might find the logic immediate, but most people will need to think about it for a moment to see how this works (I'm in that camp).

The four options above work… but good luck explaining them to your boss.

Option 2: Parity Check

You can check that Tarzan is not inside quotes by checking that it is not followed by one quote followed by an even number of quotes. That's a bit of a hack.

```
Tarzan(?!"(?:(?:[^"]*"){2})*[^"]*)
```

Simple, right? Er… Not really. There's plenty of room to introduce bugs here. And indeed, this regex will not properly handle *"Jane and Tarzan"*, where we would like *Tarzan* to match (you could get around this with a lookbehind and an alternation). In contrast, the solution to *don't match "Tarzan"* that uses the regex trick on this page will be hauntingly simple.

Option 3: The Two- or Three-Step Dance (Replace before Matching)
I'll expand on this option below when we look at cases more complex than *"Tarzan"*. In the meantime, here is the idea:

1. Replace all instances of the bad string (here *"Tarzan"*). If you're just trying to match, your replacement can be *""* (you can remove the string). If you want to replace the good strings but leave the bad strings, replace the bad strings with something distinctive, such as *"T~a~r~z~a~n"*
2. Simply match or replace the string you want (here *Tarzan*), which is now safe to do as you know that all the bad strings have been neutralized.
3. If you are replacing rather than simply matching, there is one more step: you now need to revert the distinctive strings (*"T~a~r~z~a~n"*) to their original form.

When you're working with a text editor and want to perform replacements, this is often your best bet. The technique on this page is for when you are working in a programming language that allows you to inspect your Group 1 captures, so it won't help you in EditPad Pro or Notepad++.

Option 4 for Perl, PCRE, Ruby, Python: \K
I'll also expand on this option below when we look at more complex cases than *"Tarzan"*. This option works in Perl, PCRE (C, PHP, R, …), Ruby 1.9+ and Python's alternate *regex* engine. In these regex flavors, the \K token tells the engine to discard

the characters matched up to its appearance when preparing the overall match.

We can use this feature to match unwanted content (here *"Tarzan"* and other characters that are not *Tarzan*) up to the very point where a wanted string begins (here *Tarzan*). At that point, the \K discards the unwanted content, and the engine proceeds to match the content we really want.

This solution looks like this:

```
(?:"Tarzan".*?)*\KTarzan
```

This is a compact option if you use the engines that support it, but if you're aware of explosive quantifiers, the double star will make you cringe. And it's not as compact as the solution to *don't match "Tarzan"* that you'll see below.

Okay, that was the simple case. Here the context to avoid had a fixed width: a single double-quote character on either side of the word *Tarzan*. Now let's look at the general case, where the content to exclude has a variable width.

B. General Case: variable-width exclusion (for instance between tags)
More often than not, the context we want to exclude has a width we cannot predict. For instance, suppose we want to avoid matching the string *Tarzan* somewhere between *[a]* tags, as in *[a class="variable"]Jane and Tarzan[/a]*. Not only will the string between the tags be variable (here *Jane and Tarzan*), but the tag itself may also vary, as in *[a class="variable"]*.

In such situations, you often see the big guns come out.

Option 1: Variable-Width Lookbehind
In most regex flavors, a lookbehind must have a fixed number of characters, or at least a number of characters within a specified range. However, a handful of flavors allow true variable-width lookbehinds. Among the chosen few are .NET, Matthew Barnett's alternate *regex* module for Python and JGSoft (available in RegexBuddy and EditPad).

In .NET, the question "match *Tarzan* except inside curly braces" (e.g., not in "{Jane loved Tarzan's curly hair}") can *almost* be gracefully handled with:

```
(?<!{[^}]*)Tarzan
```

Back to the Future Regex II
Why almost? Because "Tarzan" should be allowed in *{ Jane and Tarzan...*, where the left brace is left open. To check both sides, we'll need to inject a positive lookahead inside the lookbehind—stepping into *Back to the Future* territory—to assert that after we've found what we were looking for behind *Tarzan*, we can find *Tarzan* (surprise) and optional characters up to a closing curly brace. Since we're inside a negative lookbehind, this whole package is what we're trying to avoid. This is the adult version of our earlier *Back to the Future Regex I*, and it looks like this:

```
(?<!{[^}]*?(?=Tarzan[^{}]*}))Tarzan
```

What if you need more restrictions—such as also forbidding Tarzan from appearing in `[i][/i]` tags inside of `[p][/p]` tags? Yes, you can add more variable-length lookbehinds. Good luck to you as the restrictions become more numerous and complex.

Also, if the pattern to be matched is more complex than the literal *Tarzan*, the expression can fast become unmanageable.

And in Java, PHP, Ruby and Python's *re* module, you can forget about this technique altogether because infinite-width lookbehinds do not exist in these flavors.

Option 2: The Two- or Three-Step Dance (Replace before Matching)
To match all instances of *Tarzan* unless they are embedded in a string inside curly braces, one fairly heavy but simple solution is to perform a two-step dance: Replace then Match.

If we also want to **replace** all these matches, we need a third-step: a final replacement.

Step 1: You positively match all instances of *Tarzan* embedded in curly braces. If you're just trying to match, your replacement can be *""* (you can remove the string). If you want to replace the good strings but leave the bad strings, replace the word *Tarzan* with something distinctive, such as *"T~a~r~z~a~n"*. To perform the match, this simple regex would do:

```
({[^{}]*?)(Tarzan)([^}]*})
```

The string is captured into three groups: the beginning, *Tarzan*, and the end. If you're removing the bad strings before matching, your replacement would be `\1\3` or `$1$3` depending on your regex flavor. If you're replacing the bad strings before replacing the good strings, your replacement would be `\1T~a~r~z~a~n\3` or `$1T~a~r~z~a~n$3`.

If there are other contexts in which you want to avoid matching *Tarzan*, you probably have to repeat Step 1, as attempting to match all the bad strings in one big regex is fraught with risk.

Step 2: All the unwanted instances of *Tarzan* have been neutralized, so you can now match *Tarzan* without worrying about context. I realize that matching *Tarzan* in a vacuum is not that interesting. In real life you might be looking for *Tarzan* and the phone number that follows.

Optional Step 3: If the point of Step 2 was not only to match but also to perform a replacement on the acceptable *Tarzan* strings, then once that replacement is made we also need to turn all the *T~a~r~z~a~n* strings back into *Tarzan*, which is easily accomplished.

Option 3 for Perl, PCRE, Ruby and Python: \K
This option works in Perl, PCRE (C, PHP, R, …), Ruby 1.9+ and Python's alternate *regex* engine. In these engines, the \K token causes the engine to drop all it has matched up to the \K from the overall match it returns. This opens a strategy for us: we can (i) match any unwanted content (if present) up to the beginning of a wanted *Tarzan* instance, (ii) throw away that portion of the match using \K, then (iii) match *Tarzan*. This option could look like this:

```
(?:(?>{[^}]*?})[^{}]*?)*\KTarzan
```

Note that while we try to match unwanted content, we swallow entire sets of *{strings in curly braces}* without bothering to check if they contain Tarzan. We do not need to care, because we know that if something is inside curly braces, we don't want it.

Compared with the other options we've seen so far, this is fairly economical. But if you need to add conditions in which *Tarzan* cannot be matched, it can become very hard to manage.

Besides, it's still too much work compared with… (*drum roll…*)

The Best Regex Trick Ever (at last!)

If you've read up to here, well done! Without further ado, let's plunge into this technique I have been relentlessly selling you. I'm hopeful that this won't be counter-climactic in the least.

One key to this technique, a key to which I'll return several times, is that we completely disregard the overall matches returned by the regex engine: that's the trash bin. Instead, we inspect the Group 1 matches, which, when set, contain what we are looking for.

This means that you may have to write one or two extra lines of code, but that is a light price to pay for a much cleaner, lighter and easier to maintain regex. Code samples for the six typical situations are provided below.

An example is worth a picture-and-a-half, so let's revisit our first example.

Match *Tarzan* but not *"Tarzan"*

You remember the simple case where we tried to match all instances of *Tarzan* except those enclosed in double quotes? It turned out to yield solutions in varying shades of obscure, such as

`((?<=")?)Tarzan(?(1)(?!"))` and
`Tarzan(?!"(?:(?:[^"]*"){2})+[^"]*?(?:$|[\r\n]))` and
`(?:"Tarzan".*?)*\KTarzan`
Well, you'll now see how simple the problem becomes when you use the best regex trick ever:

```
"Tarzan"|(Tarzan)
```

Really? That's it?

Yes. The trick is that we match what we don't want on the left side of the alternation (the |), then we capture what we *do* want on the right side. When our programming language returns the results, we ignore the overall matches (that's the trash bin) and instead turn our whole attention to Group 1 matches, which contain what we were after.

Adding exclusions is a breeze
When there's another context we want to exclude, we simply add it as an alternation on the left, where we match it in order to neutralize it—if it's matched, it's in the trash. For instance, if we also had to exclude *Tarzan* in *Tarzania* and --*Tarzan*--, our regex would become:

```
Tarzania|--Tarzan--|"Tarzan"|(Tarzan)
```

Adding exclusions is a breeze, isn't it?

Again, the only instances of *Tarzan* we care about will be those captured by Group 1.

How Does the Technique Work?

This is simple, but it may not be entirely intuitive, so it's worth reviewing how the regex engine handles this pattern. If you feel very confident you understand the mechanics of this match, feel free to skip to the next section, the Technique in Pseudo-Regex.

A quick refresher about the regex engine
Remember that the engine has two "reading heads" which both move from left to right: one moves in the string, one moves in the regex pattern. The main thing to understand is that at the start, with the string reading head at the very beginning of the string, the engine tries to match the entire pattern at that position. If that fails, the string reading head advances by one character, and the engine again tries to match the entire pattern. Thus the engine can advance in the string one character at a time, and at each of these characters attempt an overall match, and fail, until at one starting point in the string, perhaps, an overall match is returned. Let's look at this in more detail.

When we fire up the engine, both reading heads are at the very left. At the string reading head's current position (i.e. the very left), the engine attempts to match the entire pattern. To do so, it tries to match the pattern's first token against the string's first character. If that fails, the engine's string reading head advances to the position immediately past the first character (i.e. between the first and the second character), and the pattern reading head resets to the very left. At that position, the engine once again attempts to match the whole pattern. At that stage, if the first token matches the second character, both reading heads advance, and the engine tries to match the second token against the next character.

Of course some tokens have quantifiers and match multiple characters; and within a match attempt from a given starting position in the string, the pattern reading heads often have to backtrack. But the principle remains the same: if an overall match fails, then the string reading head moves to the next position, the pattern reading head resets to the very left, and the engine once again attempts an entire match.

In "multiple matches mode", when the engine succeeds in matching the entire pattern, it records the current match, then attempts the next match starting from the position that immediately follows the last character that was just included in the match.

A Walk-Through
So let's say we are trying the original pattern `"Tarzan"|(Tarzan)` against this string:

Now Tarzan says to Jane: "Tarzan".

1. The engine's string reading head positions itself at the head of the string, before the "N" in "Now". At this position, the engine attempts to match the entire pattern `"Tarzan"|(Tarzan)`

2. At this position, the engine is unable to match the opening double quote in *"Tarzan"* because the next character is "N", so the left side of the alternation immediately fails. The engine's pattern reading head then jumps to the right side of the alternation and tries to match the initial *T* in *Tarzan*, but fails, again because the next character in the string is "N".

3. At this position in the string, the match has failed. The string reading head advances one character in the string (positioning itself between the "N" and the "o" in "Now"), and the pattern reading head resets to the very left. At this new position, the engine again attempts to match then entire pattern `"Tarzan"|(Tarzan)`

4. At this position, the engine is unable to match the opening double quote in *"Tarzan"* because the next character is "o". Likewise, the right side of the alternation fails because "T" is not "o".

5. The string reading head again advances in the string and attempts two matches that fail, the first before the "w" in "Now", the second before the space character preceding "Tarzan". The string reading head then advances in the string to the position preceding the *T*.

6. The left side of the alternation fails because the next character is not a double quote. The pattern reading head jumps to right side of the alternation, and the engine is able to match the *T*. The string reading head advances by one character, the pattern reading head advances by one token. The engine is able to match *a*, then, as the reading heads continue to advance in parallel, the engine matches *r, z, a* and *n*. The match succeeds, *Tarzan* is added to the list of matches, and since it was in parentheses it is also recorded as the Group 1 capture for this match.

7. The string reading head advances to the position after the "n" in the initial *Tarzan*, and the pattern reading head resets to the very left. At this position the engine starts a new match attempt, and fails. The string reading head advances to each position in "says to Jane: ", and as it does so, at each position the engine attempts a new match, and fails. The string reading head then advances to the position preceding the first double quote.

8. At this position, before the opening double quote, the engine attempts to match a double quote and succeeds. The string reading head advances by one character, the pattern reading head advances by one token. The engine matches the *T*, and both reading heads keep advancing in parallel until all the characters in *"Tarzan"* have been matched.

9. The match succeeds, *"Tarzan"* is added to the list of matches, but it is not captured in any capturing group as it was not surrounded by parentheses.

10. The engine returns two matches: *Tarzan* and *"Tarzan"*. We don't pay attention to the matches, but for each match we look at capturing Group 1 using our programming language. (You'll see code samples in several languages below.) For the first match, we have a non-empty capturing Group 1: *Tarzan*. That is what we were after.

The Technique in Pseudo-Regex

Here is the recipe in "pseudo-regex":

```
NotThis|NotThat|GoAway|(WeWantThis)
```
This is a game of good cop / bad cop.

Bad string
As in any good cop / bad cop routine, the bad cop comes in first. The idea is to use a series of alternations on the left to specify the contexts we want to exclude. By doing so, we force the engine to match these "bad strings". We won't even look at the overall matches—think of the set of overall matches as a garbage bin. After matching a bad string, the engine attempts the next overall match starting at the string position that immediately follows the bad string. In effect, that bad string has been skipped: this is how we manage to exclude unwanted context.

Good string
When the engine starts a match attempt at the beginning of a "good string", it can safely match it, because we know that if that string had been embedded in context we want to exclude… the engine would already have matched it and placed it in the garbage bin! Since we do match the good strings, they too go in the garbage bin. The difference is that by using capturing parentheses when we match the good strings, we capture them into Group 1.

One or two lines of code
In our code, we'll only examine these Group 1 captures. Examining Group 1 may take one or two more lines of code than examining "Group 0" (the overall matches), but that's a small price to pay for a regex that is crystal-clear and extremely easy to maintain. The code samples lower in the page will show you how to use this technique in a variety of languages for the six most common regex tasks: (i) checking if there is a match, (ii) counting matches, (iii) retrieving the first match, (iv) retrieving all matches, (v) replacing, and (vi) splitting.

This is a simple but extremely potent regex technique, don't you think?

One small thing to look out for

There are not many *bewares* with this technique, but there is one small thing to look out for. It may sound obvious, but do make sure that the expression in `(GetThis)` is not so broad that it can swallow strings that contain bad strings—specifically, strings that start one or more characters before a bad string.

For instance, suppose you want to match all words that are not inside an *<img>* tag. Let's apply our `NotThis|(GetThis)` recipe.

1. Your *NotThis* rule could look like this: `<img[^>]+>`

2. What about the *GetThis* rule? Don't use a dot-star, as on the right side of the alternation in `<img[^>]+>|(.*)`

Why not? The engine starts a match attempt at the beginning of the string. First, it tries a `<` against the first character. Say the first character is "S": the `<` fails to match. The string reading head stays at the start of the string, but the pattern reading head now moves to the right side of the alternation. The engine tries the `.*` … and the naughty dot-star swallows the "S" and the rest of the string, exclusions and all.

We are relying on the exclusion rules to remove unwanted context. But on the *GetThis* side, you can't have an expression that swallows the same context you are trying to remove! That stands to reason, but it needs to be said—and seen. I sometimes mess this up when building expressions fast, and it's good to be able to instantly spot what is going wrong.

Note that the problem only arises if the *GetThis* regex is able to match one or more characters *before* it matches a bad string. That is because at a string position that precedes a bad string by one or more characters, the exclusion rule is not able to fire, and the engine switches over to the hungry *GetThis*.

On the other hand, it is perfectly acceptable for the *GetThis* expression to have the *potential* to match a bad string, as long as it only has that potential at the very start of a bad string. Why? Because this potential never has a chance to come to fruition. Since the exclusion regex patterns are on the left of the alternation, these patterns neutralize bad strings before the *GetThis* regex can ever get to them.

In our example, this regex would do the job: `<img[^>]+>|(\w+)`

More Applications of the Technique

Let's now explore other examples using the technique. At the very end, we'll also look at a neat variation for Perl and the PCRE engine (which PHP and Apache use).

Match *Tarzan* but not in *{Tarzan's curly braces}*

Remember how complex the typical case was before? The task was to match *Tarzan*, except when it lives somewhere between curly braces.

Now all we have to do is apply our recipe:

```
Not_this_context|(WeWantThis)
```

Okay, first off, we know that *(WeWantThis)* is simply `(Tarzan)`.

Now how can we express *Not_this_context*? The unwanted context is *Tarzan* inside curly braces. Delightfully, for this, we use something as compact as `{[^}]*}`, and I'll explain why in a short moment. This small expression simply matches the entire content of a pair of curly braces. For this example, we're assuming that braces are {never {nested}}.

This gives us:

```
{[^}]*}|(Tarzan)
```

All we have to do is retrieve the matches from Group 1. Too easy!! Of course in real life we would probably not look just for the word *Tarzan*, but for some variable content, such as `Tarzan\d+`

Please skip it simple!
Please note this trick within a trick: to specify the exclusion rule, we did not bother to write a whole expression to match *Tarzan* inside curly braces, such as:

```
{[^}]*?Tarzan[^}]*}
```

Instead, we just matched the content of *any* curly braces:

```
{[^}]*}
```

Why? Because if something is inside curly braces, we know that we don't want anything to do with it. So we can go ahead and skip all sets of curly braces without bothering to look inside!

This is what I call "skipping it simple".

Now let's take it up a notch.

Match *Tarzan* but not in contexts A, B and C

Your boss just told you that not only do we want to avoid *Tarzan* inside curly braces, we also want to leave the muscular vine hopper in his jungle when he appears within sections that start with *BEGIN* and end with *END*. Also, sentences starting with "Therefore" are definitely excluded.

How is that for a change of specs? Is she trying to make you break a sweat? You must have solved the first assignment too fast. If you had done it with one of the typical techniques, at this stage you might be pulling your hair. Instead, this is what you do:

Step 1: spend 57 seconds revising the original expression to this:

```
\bBEGIN\b.*?\bEND\b|Therefore.*?[.!?]|{[^}]*}|(Tarzan)
```

Step 2: clean up your inbox for a couple of hours before announcing to your boss that it was curly, but that *by gawd…* you've wrestled that regex to the ground!

So what have we done? We've just followed the recipe and added two exclusions to the original regex in alternations at the left. The first exclusion, which could have been a simple `BEGIN.*?END`, matches any sequence starting with *BEGIN* and ending with *END*. You've added the `\b` boundaries because you're nice and you want to give your boss a real *END*, not just any old *WENDY*. The second exclusion swallows any string that starts with *Therefore* and ends with the three characters in the `[.!?]` character class—so chosen because your boss told you to assume that all sentences end with periods, question marks or bangs.

Okay, we're feeling great. What's the next use of our golden technique?

Match every word except *Tarzan*

So far, we've been looking at questions of the form:

> Match X unless it is in contexts a, b and c.

Now let's look at a family of questions that sound quite different but reduce to the same:

> Match every word except words a, b and c.

To start easy, let's try to match every word except *Tarzan*. Hey, that's simple:

```
\bTarzan\b|(\w+)
```

By the way, this is an interesting case because by itself, the `\w+` would be able to match *Tarzan*. However, it is never able to fire in that situation, because by the time we get to an instance of *Tarzan*, the exclusion rule has already matched it. This is explained in more detail in the section about one small thing to look out for.

Note also that as it is, the regex will capture *antiTarzan* and *Tarzania*. That's a feature, not a bug (see the `\b` boundaries.)

Let's take it up a notch and talk about blacklists, a commonly requested regex task.

Match every word except those on a blacklist

This time we want to blacklist the words *Tarzan*, *Jane* and *Superman*. Hey, no problem. We add exclusions on the left, and our regex becomes:

```
\bTarzan\b|\bJane\b|\bSuperman\b|(\w+)
```

or, more gracefully:

```
\b(?:Tarzan|Jane|Superman)\b|(\w+)
```

You can try it online with "Tarzan, Jane and Superman hopped from vine to vine." Remember that what we're looking at is the Group 1 matches, which are shown in the lower right-hand panel and highlighted differently from the plain matches.

Let's now talk about an application of the technique which, to untrained ears, sounds completely different:

Ignore Content of This Kind

Sometimes someone may present you with a regex problem and phrase it in this manner:

> I want to ignore A.

It's useful to notice that this wording is just a variation on

> Match everything except A

Didn't we just see that one? We did. Even so, let's stay sharp by practicing one more time, using this assignment: *ignore bolded content*.

Maybe you can convince your boss to reword this as "match all content except anything in bold". By "in bold", let's say we're talking about content within <*b*> tags. And by "content", let's say we're talking about sequences of word and whitespace characters.

Using our recipe, we can translate the assignment like so:

```
<b>[^<]*</b>|([\w\s]+)
```

As a reminder (see the lookout section for details), it would not do to use a `(.*)` in the *GetThis* section, because at any point in the string prior to a bolded section, the exclusion rule would fail, while the naughty dot-star would swallow the entire string from that point to the end—including any bolded sections.

In that case, how about the lazy quantifier `(.*?)`, you might wonder? You could do that—but make sure to see the section explaining why lazy quantifiers are expensive on the *Mastering Quantifiers* page.

Back the the article's Table of Contents

A Variation: Deleting the Matches

Sometimes, you want to match content in order to delete it. In this case, there is a simple tweak to our usual recipe that allows us to delete the matches directly without inspecting Group 1 captures. To search, instead of our usual recipe:

```
NotThis|NotThat|GoAway|(WeWantThis)
```

We use:

```
(KeepThis|KeepThat|KeepTheOther)|DeleteThis
```

As you can see, the location of the parentheses has been inverted. We can now replace the match with Group 1. There are two cases:
- If the match took place on the left branch of the alternation, and therefore captured to Group 1, the match is replaced with itself (no change);
- If the match took place on the right side of the alternation, the match is replaced with Group 1, which is empty: it is therefore deleted.

Here is an interesting variation to do the same:

```
(KeepThis)|(KeepThat)|(KeepTheOther)|DeleteThis
```

For the replacement, we concatenate Groups 1, 2 and 3 (in any order). Since only one of those groups is ever captured (if any), the other two groups contain empty strings. Once again, the match is replaced with itself (if captured) or with an empty string.

There is no standard for replacement syntax, so in one language this may look like \1\2\3, $1$2$3 or m.group(1) + m.group(2) + m.group(3).

Variation for Perl, PCRE and Python: (*SKIP)(*FAIL)

Perl, PCRE (C, PHP, R…) and Python's alternate *regex* engine have a variation that uses almost entirely the same syntax, but that returns the desired matches as the overall match instead of returning them in capture Group 1. In these flavors, this is a neat trick to know as it can save us one or two lines of code.

Remember that in our technique, when we express a series of unwanted contexts in alternations to be matched and thrown in the garbage bin, such as NotThis|NotThat, the key to success is that when such undesirable areas of the strings are matched, they are in effect SKIPPED. After matching them, the engine attempts the next match starting at the position immediately following the preceding match. The entire area to be excluded has been gobbled up, and therefore skipped.

Well, with Perl, PCRE and Python's alternate *regex* engine, you can use a construct that makes the engine to match that undesirable content, then fail the match… after which the engine skips the entire substring that just failed and starts the next match attempt at the position immediately following the bad string. This allows us to do the same as we've been doing, but we no longer need parentheses to capture the content we want because there is no longer a garbage bin full of unwanted matches to be ignored. We can inspect the matches directly, because the pattern only matches what we want.

That syntax can either be written as (*SKIP)(*FAIL), (*SKIP)(*F) or (*SKIP)(?!). That's because (*FAIL) and (*F) are both synonyms for (?!), which, as we saw on the tricks page, is an expression that never matches, forcing the engine to backtrack in search of a different match.

As for (*SKIP), it's a *backtracking control verb* in Perl, PCRE and Python's alternate *regex* engine. You can read all about it on my page about backtracking control verbs. When the engine tries to backtrack across (*SKIP), the match attempt explodes. Instead of starting the next match attempt at the next starting position in the string, the engine advances to the string position corresponding to where (*SKIP) was encountered. This means that anything to the left of (*SKIP) is never visited again. Apart from time-saving benefits, this technique allows us to reject entire chunks of text in one go.

Remember the overall recipe to avoid context X? It was

```
Not_X|(GetThis)
```

Using Perl, PCRE (PHP, R, C…) or Python's alternate *regex* engine, we can accomplish the same with either of these:

```
Not_A(*SKIP)(*FAIL)|GetThis Not_A(*SKIP)(*F)|GetThis Not_A(*SKIP)(?!)|GetThis
```

Note that the parentheses around *GetThis* have disappeared. Whenever the engine is able to match *Not_A*, the (*SKIP)(*FAIL) construct causes it to reject that entire chunk of text and start the next match attempt immediately afterwards. Whenever the engine is **not** able to match *Not_A*, it jumps to the right branch of the alternation | and tries to match *GetThis*. If this fails, the engine starts the next match attempt at the next starting position in the subject text, as always.

If we want to avoid three contexts A, B and C, our technique used to do this: `Not_A|Not_B|Not_C|(GetThis)`
In Perl and PHP, we can instead say something like one of these:

```
Not_A(*SKIP)(*FAIL)|Not_B(*SKIP)(*F)|Not_C(*SKIP)(?!)|GetThis
(?:Not_A|Not_B|Not_C)(*SKIP)(*FAIL)|(GetThis)
```

Back the the article's Table of Contents

Code Samples

To complete this article, I'd like to provide a full implementation in several common languages.

A Call to Help
May 2014. I'm calling for your help to translate the examples provided to languages in which you are fluent (see code translators needed). **In advance, thank you.**

The six tasks performed by the code samples

The code performs the six most common regex tasks. The first four tasks answer the most common questions we use regex for:

✽ Does the string match?
✽ How many matches are there?
✽ What is the first match?
✽ What are all the matches?

The last two tasks perform two other common regex tasks:

✽ Replace all matches
✽ Split the string

Learn a new engine!
The code samples should allow even complete beginners to pick code fragments that suit their needs and tweak them to their liking.

Please rest assured that *beginner* is not a condescending term here, and I am expecting "advanced beginners" to take advantage of the code. If you are proficient in regex in the context of one programming language, you may be curious to test out other engines, but also worried about the learning curve. Apart from illustrating various uses of the technique, the code samples allow you to start experimenting in a variety of regex flavors.

The assignment for the code samples
All the code samples tackle the same assignment. Our assignment is to match *Tarzan* followed by any number of digits, for instance *Tarzan111*, **except:**

1. between quotes, as in *"Tarzan123"*,
2. somewhere inside curly braces, as in *{ Jane Tarzan123 }*

For this assignment, I will use `\d` without attempting to distinguish between ASCII digits and Unicode digits, as that is not the point of the exercise. Just be aware that in some engines `\d` only matches the ASCII digits 0 to 9, while in others it also matches digits in other alphabets. If you want to be consistent, use `[0-9]`

The Test Strings
To test the code, we'll use one string that produces two matches and a small variation that should produce none.

1. The string below should produce two matches: *Tarzan11* and *Tarzan22*

```
Jane" "Tarzan12" Tarzan11@Tarzan22 {4 Tarzan34}
```

2. To test failure cases, I suggest you capitalize two *z* characters as in the string below, which should produce no matches:
Jane" "Tarzan12" TarZan11@TarZan22 {4 Tarzan34}

The Regex
Here is the regex we'll use:
`{[^}]+}|"Tarzan\d+"|(Tarzan\d+)`
1. The first part of the alternation `{[^}]+}` matches and neutralizes any content between curly quotes.

2. The second part of the alternation `"Tarzan\d+"` matches and neutralizes instances where the sought string is embedded within double quotes. You may ask why I didn't simply neutralize *any* content between double quotes in similar fashion to the first part of the alternation, using `"[^"]+"`. For most strings, that would have worked, but if you carefully inspect the test string, you'll see that I sneaked in an extra double quote after *Jane*. I did so to illustrate a safe regex work practice. See, if for any reason the subject string has an odd number of double quotes as is the case here, you cannot be sure that two quotes matched by `"[^"]+"` belong together. Indeed, for our test string, that code would match a single space within double

quotes, and the regex would (wrongly) capture *Tarzan12* into Group 1. Therefore, when working with quotes, being specific as in `"Tarzan\d+"` is safer. In the case of braces (where there are distinct characters for the left and right sides), the risk of mismatches is far lower.

3. The third part of the alternation `(Tarzan\d+)` matches *Tarzan* and the following digits and captures the match into Group 1.

Here are jump points to code samples in various languages.

Implemented
PHP
C#
Python
Java
JavaScript
Ruby
Perl
VB.NET

Not Yet Implemented
Visual C++
Scala
Other language of your choice

PHP Code Sample

For PHP, I'll provide two samples. The first illustrates the main technique on this page. The second illustrates the *(*SKIP)(*F)* variation specific to Perl and PHP, which is a little lighter.

Sample #1: The Core Technique
If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Sample #2: The (*SKIP)(*F) Variation
This sample implements the technique explained in the Variation for Perl and PCRE section.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

C# Code Sample

If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

Python Code Sample

If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

Java Code Sample

If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

JavaScript Code Sample

If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

Ruby Code Sample

If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

Perl Code Sample

If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
or leave the site to view an online demo

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

VB.NET Code Sample

If you see ways to improve the code, please leave a comment.

Please note that usually you will choose to perform **only one** of the six tasks in the code, so your own code will be much shorter.

Click to Show / Hide code
(The code compiles perfectly in VS2015, but no online demo supplied
because the VB.NET in ideone chokes on anonymous functions.)

Back to the Code Samples explanation and languages
Back the the article's Table of Contents

Code Translators Needed

I would love to enlist your help so the page can provide working code in more languages. Please see the list of languages for languages currently implemented and missing.

If you wish, you will be duly acknowledged with your name or an alias of your choice.

Are you willing to help? Fantastic. To make things easy for me, your code needs to mirror the specs of the other samples. Here are the requirements that come to mind:

✽ **Completeness.** The idea is to provide code that someone who has never used your language is able to plug in to an IDE, compile (if needed) and run. So please include any opening braces and the few needed lines to import any relevant libraries.

✽ **Conciseness.** By the same token, please ommit any unneeded fluff, such as unneeded libraries and classes.

✽ **Same example.** To keep things consistent, please use the regex and subject string provided.

✽ **Six tasks.** The code must include separate sections that could be run separately if needed, addressing the four common tasks illustrated by the code already on the page: (i) checking whether there is a match, (ii) counting the matches, (iii) returning the first match, (iv) returning all matches, (v) replacing all matches, (vi) splitting the string.

✽ **Formatted output.** If you run the existing demos, you'll see that they output certain strings at each step to inform us of where we are in the code. Your code should output those same strings.

✽ **Link to a working demo.** For consistency, if ideone.com supports your language, please use it.

If you paste your code in the comment form it may not make it to me intact, but you can paste an ideone.com link or a brief message. I'll reply. Html won't work in the comment form.

A million thanks in advance!

Well, I think that's about all I have to say about this technique at the moment. Writing it was a big journey. I hope you had a blast reading it.

Wishing you loads of fun on your travels in regexland,

Rex

At this stage you might like to treat yourself to some

Regex Humor

…or just visit the next page.

NEXT»

Regex Cookbook

Ask Rex

Leave a Comment
1-7 of 7 Threads
Ivan
July 05, 2020 - 15:16
Subject: An error in the JavaScript Code Sample

Hi,
Line 37 of the JS code,
"if (group1 == "") return m;"
should be
"if (group1 == undefined) return m;"
for the code to work correctly.
Reply to Ivan
Rex
July 05, 2020 - 21:11
Subject: RE: An error in the JavaScript Code Sample

Thank you Ivan. The code worked when I wrote it, but JS specs change over time and vary from platform to platform, so I'm glad you let me know about the latest. Warm regards, Rex
Rex
September 29, 2015 - 10:42
Subject: RE: nitpick

Hi Toomas, Thank you so much for your nitpicks, man! I really appreciate them. Perl is not my idiom so I'm sure what I have is quite heavy. Added your Perl code as a comment line above what was there. Fixed the others. Wishing you a fun week, Rex
Rex
September 27, 2015 - 12:30
Subject: RE: Typo

Hi Omer, Thank you very much for reporting typos. I really appreciate it. Fixed. Wishing you a fun weekend, Rex
Omer – Earth
September 27, 2015 - 11:54
Subject: Typo

First sentence after option 3 heading: s/that/than/
Joel
September 27, 2015 - 04:20
Subject: Excellent article - thanks for the regex help!

Very well done. All your step-by-step examples make this article superb.
Lane
July 07, 2014 - 09:15
Subject: When the simple answer is the most profound

That is so simple that it's genius! I just started to learn regex and am glad I found this site so I don't waste time struggling with it when you cut right to the chase. Thanks!
Joe – Texas
June 03, 2014 - 13:14
Subject: Thank you!

You are awesome. Thanks for the trick and the page.