Regular Expression Tutorial
Learn How to Use and Get The Most out of Regular Expressions

In this tutorial, I will teach you all you need to know to be able to craft powerful time-saving regular expressions. I will start with the most basic concepts, so that you can follow this tutorial even if you know nothing at all about regular expressions yet.

But I will not stop there. I will also explain how a regular expression engine works on the inside, and alert you at the consequences. This will help you to understand quickly why a particular regex does not do what you initially expected. It will save you lots of guesswork and head-scratching when you need to write more complex regexes.

What Regular Expressions Are Exactly - Terminology

Basically, a regular expression is a pattern describing a certain amount of text. Their name comes from the mathematical theory on which they are based. But we will not dig into that. Since most people including myself are lazy to type, you will usually find the name abbreviated to regex or regexp. I prefer regex, because it is easy to pronounce the plural "regexes". On this web site, regular expressions are printed as `regex`. If your browser has proper support for cascading style sheets, the regex should be highlighted in red.

This first example is actually a perfectly valid regex. It is the most basic pattern, simply matching the literal text `regex`. A "match" is the piece of text, or sequence of bytes or characters that pattern was found to correspond to by the regex processing software. Matches are highlighted in blue on this site.

`\b[A-Z0-9._%-]+@[A-Z0-9._%-]+\.[A-Z0-9._%-]{2,4}\b` is a more complex pattern. It describes a series of letters, digits, dots, percentage signs and underscores, followed by an at sign, followed by another series of letters, digits, dots, percentage signs and underscores, finally followed by a single dot and between two and four letters. In other words: this pattern describes an email address.

With the above regular expression pattern, you can search through a text file to find email addresses, or verify if a given string looks like an email address. In this tutorial, I will use the term "string" to indicate the text that I am applying the regular expression to. I will highlight them in `green`. The term "string" or "character string" is used by programmers to indicate a sequence of characters. In practice, you can use regular expressions with whatever data you can access using the application or programming language you are working with.

Different Regular Expression Engines

A regular expression "engine" is a piece of software that can process regular expressions, trying to match the pattern to the given string. Usually, the engine is part of a larger application and you do

not access the engine directly. Rather, the application will invoke it for you when needed, making sure the right regular expression is applied to the right file or data.

As usual in the software world, different regular expression engines are not fully compatible with each other. It is not possible to describe every kind of engine and regular expression syntax (or "flavor") in this tutorial. I will focus on the regex flavor used by Perl 5, for the simple reason that this regex flavor is the most popular one, and deservedly so. Many more recent regex engines are very similar, but not identical, to the one of Perl 5. Examples are the open source PCRE engine (used in many tools and languages like PHP), the .NET regular expression library, and the regular expression package included with version 1.4 and later of the Java JDK. I will point out to you whenever differences in regex flavors are important, and which features are specific to the Perl-derivatives mentioned above.

Give Regexes a First Try

You can easily try the following yourself in a text editor that supports regular expressions, such as EditPad Pro. If you do not have such an editor, you can download the free evaluation version of EditPad Pro to try this out. EditPad Pro's regex engine is fully functional in the demo version. As a quick test, copy and paste the text of this page into EditPad Pro. Then select Edit|Search and Replace from the menu. In the search pane that appears near the bottom, type in `regex` in the box labeled "Search Text". Mark the "Regular expression" checkbox, unmark "All open documents" and mark "Start from beginning". Then click the Search button and see how EditPad Pro's regex engine finds the first match. When "Start from beginning" is checked, EditPad Pro uses the entire file as the string to try to match the regex to.

When the regex has been matched, EditPad Pro will automatically turn off "Start from beginning". When you click the Search button again, the remainder of the file, after the highlighted match, is used as the string. When the regex can no longer match the remaining text, you will be notified, and "Start from beginning" is automatically turned on again.

Now try to search using the regex `reg(ular expressions?|ex(p|es)?)`. This regex will find all names, singular and plural, I have used on this page to say "regex". If we only had plain text search, we would have needed 5 searches. With regexes, we need just one search. Regexes save you time when using a tool like EditPad Pro.

If you are a programmer, your software will run faster since even a simple regex engine applying the above regex once will outperform a state of the art plain text search algorithm searching through the data five times. Regular expressions also reduce development time. With a regex engine, it takes only one line (e.g. in Perl, PHP, Java or .NET) or a couple of lines (e.g. in C using PCRE) of code to, say, check if the user's input looks like a valid email address.

Literal Characters

The most basic regular expression consists of a single literal character, e.g.: `a`. It will match the first occurrence of that character in the string. If the string is `Jack is a boy`, it will match the `a` after the `J`. The fact that this `a` is in the middle of the word does not matter to the regex engine. If it

matters to you, you will need to tell that to the regex engine by using word boundaries. We will get to that later.

This regex can match the second `a` too. It will only do so when you tell the regex engine to start searching through the string after the first match. In a text editor, you can do so by using its "Find Next" or "Search Forward" function. In a programming language, there is usually a separate function that you can call to continue searching through the string after the previous match.

Similarly, the regex `cat` will match `cat` in `About cats and dogs`. This regular expression consists of a series of three literal characters. This is like saying to the regex engine: find a `c`, immediately followed by an `a`, immediately followed by a `t`.

Note that regex engines are case sensitive by default. `cat` does not match `Cat`, unless you tell the regex engine to ignore differences in case.

Special Characters

Because we want to do more than simply search for literal pieces of text, we need to reserve certain characters for special use. In the regex flavors discussed in this tutorial, there are 11 characters with special meanings: the opening square bracket `[`, the backslash `\`, the caret `^`, the dollar sign `$`, the period or dot `.`, the vertical bar or pipe symbol `|`, the question mark `?`, the asterisk or star `*`, the plus sign `+`, the opening round bracket `(` and the closing round bracket `)`. These special characters are often called "metacharacters".

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash. If you want to match `1+1=2`, the correct regex is `1\+1=2`. Otherwise, the plus sign will have a special meaning.

Note that `1+1=2`, with the backslash omitted, is a valid regex. So you will not get an error message. But it will not match `1+1=2`. It would match `111=2` in `123+111=234`, due to the special meaning of the plus character.

If you forget to escape a special character where its use is not allowed, such as in `+1`, then you will get an error message.

All other characters should not be escaped with a backslash. That is because the backslash is also a special character. The backslash in combination with a literal character can create a regex token with a special meaning. E.g. `\d` will match a single digit from 0 to 9.

Special Characters and Programming Languages

If you are a programmer, you may be surprised that characters like the single quote and double quote are not special characters. That is correct. When using a regular expression or grep tool like PowerGREP or the search function of a text editor like EditPad Pro, you should not escape or repeat the quote characters like you do in a programming language.

In your source code, you have to keep in mind which characters get special treatment inside strings by your programming language. That is because those characters will be processed by the compiler, before the regex library sees the string. So the regex `1\+1=2` must be written as `"1\\+1=2"` in C++ code. The C++ compiler will turn the escaped backslash in the source code into a single backslash in the string that is passed on to the regex library. To match `c:\temp`, you need to use the regex `c:\\temp`. As a string in C++ source code, this regex becomes `"c:\\\\temp"`. Four backslashes to match a single one indeed.

See the tools and languages section of this website for more information on how to use regular expressions in various programming languages.

Non-Printable Characters

You can use special character sequences to put non-printable characters in your regular expression. `\t` will match a tab character (ASCII 0x09), `\r` a carriage return (0x0D) and `\n` a line feed (0x0A). Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

You can include any character in your regular expression if you know its hexadecimal ASCII or ANSI code for the character set that you are working with. In the Latin-1 character set, the copyright symbol is character 0xA9. So to search for the copyright symbol, you can use `\xA9`. Another way to search for a tab is to use `\x09`. Note that the leading zero is required.

First Look at How a Regex Engine Works Internally

Knowing how the regex engine works will enable you to craft better regexes more easily. It will help you understand quickly why a particular regex does not do what you initially expected. This will save you lots of guesswork and head-scratching when you need to write more complex regexes.

There are two kinds of regular expression engines: text-directed engines, and regex-directed engines. Jeffrey Friedl calls them DFA and NFA engines, respectively. All the regex flavors treated in this tutorial are based on regex-directed engines. This is because certain very useful features, such as lazy quantifiers and backreferences, can only be implemented in regex-directed engines. No surprise that this kind of engine is more popular.

Notable tools that use text-directed engines are awk, egrep, flex, lex, MySQL and Procmail. For awk and egrep, there are a few versions of these tools that use a regex-directed engine.

You can easily find out whether the regex flavor you intend to use has a text-directed or regex-directed engine. If backreferences and/or lazy quantifiers are available, you can be certain the engine is regex-directed. You can do the test by applying the regex `regex|regex not` to the string `regex not`. If the resulting match is only `regex`, the engine is regex-directed. If the result is `regex not`, then it is text-directed. The reason behind this is that the regex-directed engine is "eager".

In this tutorial, after introducing a new regex token, I will explain step by step how the regex engine actually processes that token. This inside look may seem a bit long-winded at certain times. But

understanding how the regex engine works will enable you to use its full power and help you avoid common mistakes.

The Regex-Directed Engine Always Returns the Leftmost Match

This is a very important point to understand: a regex-directed engine will always return the leftmost match, even if a "better" match could be found later. When applying a regex to a string, the engine will start at the first character of the string. It will try all possible permutations of the regular expression at the first character. Only if all possibilities have been tried and found to fail, will the engine continue with the second character in the text. Again, it will try all possible permutations of the regex, in exactly the same order. The result is that the regex-directed engine will return the *leftmost* match.

When applying `cat` to `He captured a catfish for his cat.`, the engine will try to match the first token in the regex `c` to the first character in the match `H`. This fails. There are no other possible permutations of this regex, because it merely consists of a sequence of literal characters. So the regex engine tries to match the `c` with the `e`. This fails too, as does matching the `c` with the space. Arriving at the 4th character in the match, `c` matches `c`. The engine will then try to match the second token `a` to the 5th character, `a`. This succeeds too. But then, `t` fails to match `p`. At that point, the engine knows the regex cannot be matched starting at the 4th character in the match. So it will continue with the 5th: `a`. Again, `c` fails to match here and the engine carries on. At the 15th character in the match, `c` again matches `c`. The engine then proceeds to attempt to match the remainder of the regex at character 15 and finds that `a` matches `a` and `t` matches `t`.

The entire regular expression could be matched starting at character 15. The engine is "eager" to report a match. It will therefore report the first three letters of catfish as a valid match. The engine never proceeds beyond this point to see if there are any "better" matches. The first match is considered good enough.

In this first example of the engine's internals, our regex engine simply appears to work like a regular text search routine. A text-directed engine would have returned the same result too. However, it is important that you can follow the steps the engine takes in your mind. In following examples, the way the engine works will have a profound impact on the matches it will find. Some of the results may be surprising. But they are always logical and predetermined, once you know how the engine works.

Character Classes or Character Sets

With a "character class", also called "character set", you can tell the regex engine to match only one out of several characters. Simply place the characters you want to match between square brackets. If you want to match an a or an e, use `[ae]`. You could use this in `gr[ae]y` to match either `gray` or `grey`. Very useful if you do not know whether the document you are searching through is written in American or British English.

A character class matches only a single character. `gr[ae]y` will not match `graay`, `graey` or any such thing. The order of the characters inside a character class does not matter. The results are identical.

You can use a hyphen inside a character class to specify a range of characters. `[0-9]` matches a *single* digit between 0 and 9. You can use more than one range. `[0-9a-fA-F]` matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. `[0-9a-fxA-FX]` matches a hexadecimal digit or the letter X. Again, the order of the characters and the ranges does not matter.

Useful Applications

Find a word, even if it is misspelled, such as `sep[ae]r[ae]te` or `li[cs]en[cs]e`.

Find an identifier in a programming language with `[A-Za-z_][A-Za-z_0-9]*`.

Find a C-style hexadecimal number with `0[xX][A-Fa-f0-9]+`.

Negated Character Classes

Typing a caret after the opening square bracket will negate the character class. The result is that the character class will match any character that is *not* in the character class. Unlike the dot, negated character classes also match (invisible) line break characters.

It is important to remember that a negated character class still must match a character. `q[^u]` does *not* mean: "a q not followed by a u". It means: "a q followed by a character that is not a u". It will not match the q in the string `Iraq`. It will match the q and the space after the q in `Iraq is a country`. Indeed: the space will be part of the overall match, because it is the "character that is not a u" that is matched by the negated character class in the above regexp. If you want the regex to match the q, and only the q, in both strings, you need to use negative lookahead: `q(?!u)`. But we will get to that later.

Metacharacters Inside Character Classes

Note that the only special characters or metacharacters inside a character class are the closing bracket (]), the backslash (\), the caret (^) and the hyphen (-). The usual metacharacters are normal characters inside a character class, and do not need to be escaped by a backslash. To search for a star or plus, use `[+*]`. Your regex will work fine if you escape the regular metacharacters inside a character class, but doing so significantly reduces readability.

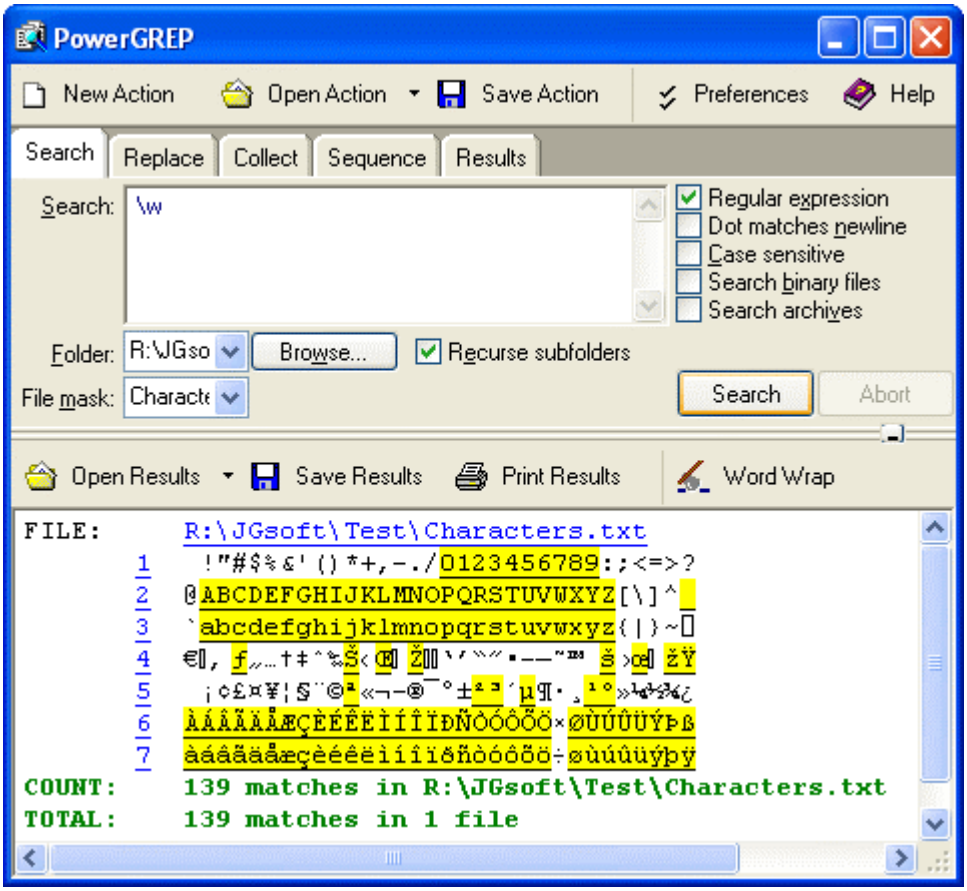
To include a backslash as a character without any special meaning inside a character class, you have to escape it with another backslash. `[\\x]` matches a backslash or an x. The closing bracket (]), the caret (^) and the hyphen (-) can be included by escaping them with a backslash, or by placing them in a position where they do not take on their special meaning. I recommend the latter

method, since it improves readability. To include a caret, place it anywhere except right after the opening bracket. `[x^]` matches an x or a caret. You can put the closing bracket right after the opening bracket, or the negating caret. `[]x]` matches a closing bracket or an x. `[^]x]` matches any character that is not a closing bracket. The hyphen can be included right after the opening bracket, or right before the closing bracket, or right after the negating caret. Both `[-x]` and `[x-]` match an x or a hyphen.

Shorthand Character Classes

Since certain character classes are used often, a series of shorthand character classes are available. `\d` is short for `[0-9]`.

`\w` stands for "word character". Exactly which characters it matches differs between regex flavors. In all flavors, it will include `[A-Za-z]`. In most, the underscore and digits are also included. In some flavors, word characters from other languages may also match. In EditPad Pro, for example, the actual character range depends on the script you have chosen in Options|Font. If you are using the Western script, characters with diacritics used in languages such as French and Spanish will be included. If you are using the Cyrillic script, Russian characters will be included, etc. The best way to find out is to do a couple of tests with the regex flavor you are using. In the screen shot, you can see the characters matched by `\w` in PowerGREP when using the Western script.



`\s` stands for "whitespace character". Again, which characters this actually includes, depends on the regex flavor. In all flavors discussed in this tutorial, it includes `[ \t]`. That is: `\s` will match a space or a tab. In most flavors, it also includes a carriage return or a line feed as in `[ \t\r\n]`. Some flavors include additional, rarely used non-printable characters such as vertical tab and form feed.

Shorthand character classes can be used both inside and outside the square brackets. `\s\d` matches a whitespace character followed by a digit. `[\s\d]` matches a single character that is either whitespace or a digit. When applied to `1 + 2 = 3`, the former regex will match ` 2` (space two), while the latter matches `1` (one). `[\da-fA-F]` matches a hexadecimal digit, and is equivalent to `[0-9a-fA-F]`.

Negated Shorthand Character Classes

The above three shorthands also have negated versions. `\D` is the same as `[^\d]`, `\W` is short for `[^\w]` and `\S` is the equivalent of `[^\s]`.

Be careful when using the negated shorthands inside square brackets. `[\D\S]` is *not* the same as `[^\d\s]`. The latter will match any character that is not a digit or whitespace. So it will match `x`, but not `8`. The former, however, will match any character that is either not a digit, or is not whitespace. Because a digit is not whitespace, and whitespace is not a digit, `[\D\S]` will match any character, digit, whitespace or otherwise.

Repeating Character Classes

If you repeat a character class by using the `?`, `*` or `+` operators, you will repeat the entire character class, and not just the character that it matched. The regex `[0-9]+` can match `837` as well as `222`.

If you want to repeat the matched character, rather than the class, you will need to use backreferences. `([0-9])\1+` will match `222` but not `837`. When applied to the string `833337`, it will match `3333` in the middle of this string. If you do not want that, you need to use lookahead and lookbehind.

But I digress. I did not yet explain how character classes work inside the regex engine. Let us take a look at that first.

Looking Inside The Regex Engine

As I already said: the order of the characters inside a character class does not matter. `gr[ae]y` will match `grey` in `Is his hair grey or gray?`, because that is the *leftmost match*. We already saw how the engine applies a regex consisting only of literal characters. Below, I will explain how it

applies a regex that has more than one permutation. That is: `gr[ae]y` can match both `gray` and `grey`.

Nothing noteworthy happens for the first twelve characters in the string. The engine will fail to match `g` at every step, and continue with the next character in the string. When the engine arrives at the 13th character, `g` is matched. The engine will then try to match the remainder of the regex with the text. The next token in the regex is the literal `r`, which matches the next character in the text. So the third token, `[ae]` is attempted at the next character in the text (`e`). The character class gives the engine two options: match `a` or match `e`. It will first attempt to match `a`, and fail.

But because we are using a regex-directed engine, it must continue trying to match all the other permutations of the regex pattern before deciding that the regex cannot be matched with the text starting at character 13. So it will continue with the other option, and find that `e` matches `e`. The last regex token is `y`, which can be matched with the following character as well. The engine has found a complete match with the text starting at character 13. It will return `grey` as the match result, and look no further. Again, the *leftmost match* was returned, even though we put the `a` first in the character class, and `gray` could have been matched in the string. But the engine simply did not get that far, because another equally valid match was found to the left of it.

The Dot Matches (Almost) Any Character

In regular expressions, the dot or period is one of the most commonly used metacharacters. Unfortunately, it is also the most commonly misused metacharacter.

The dot matches a single character, without caring what that character is. The only exception are newline characters. In all regex flavors discussed in this tutorial, the dot will *not* match a newline character by default. So by default, the dot is short for the negated character class `[^\n]` (UNIX regex flavors) or `[^\r\n]` (Windows regex flavors).

This exception exists mostly because of historic reasons. The first tools that used regular expressions were line-based. They would read a file line by line, and apply the regular expression separately to each line. The effect is that with these tools, the string could never contain newlines, so the dot could never match them.

Modern tools and languages can apply regular expressions to very large strings or even entire files. All regex flavors discussed here have an option to make the dot match all characters, including newlines. In RegexBuddy, EditPad Pro or PowerGREP, you simply tick the checkbox labeled "dot matches newline".

In Perl, the mode where the dot also matches newlines is called "single-line mode". This is a bit unfortunate, because it is easy to mix up this term with "multi-line mode". Multi-line mode only affects anchors, and single-line mode only affects the dot. You can activate single-line mode by adding an s after the regex code, like this: `m/^regex$/s;`.

Other languages and regex libraries have adopted Perl's terminology. When using the regex classes of the .NET framework, you activate this mode by specifying `RegexOptions.Singleline`, such as in `Regex.Match("string", "regex", RegexOptions.Singleline)`.

In all programming languages and regex libraries I know, activating single-line mode has no effect other than making the dot match newlines. So if you expose this option to your users, please give it a clearer label like was done in RegexBuddy, EditPad Pro and PowerGREP.

Use The Dot Sparingly

The dot is a very powerful regex metacharacter. It allows you to be lazy. Put in a dot, and everything will match just fine when you test the regex on valid data. The problem is that the regex will also match in cases where it should not match. If you are new to regular expressions, some of these cases may not be so obvious at first.

I will illustrate this with a simple example. Let's say we want to match a date in mm/dd/yy format, but we want to leave the user the choice of date separators. The quick solution is `\d\d.\d\d.\d\d`. Seems fine at first. It will match a date like `02/12/03` just fine. Trouble is: `02512703` is also considered a valid date by this regular expression. In this match, the first dot matched `5`, and the second matched `7`. Obviously not what we intended.

`\d\d[- /.]\d\d[- /.]\d\d` is a better solution. This regex allows a dash, space, dot and forward slash as date separators. Remember that the dot is not a metacharacter inside a character class, so we do not need to escape it with a backslash.

This regex is still far from perfect. It matches `99/99/99` as a valid date. `[0-1]\d[- /.][0-3]\d[- /.]\d\d` is a step ahead, though it will still match `19/39/99`. How perfect you want your regex to be depends on what you want to do with it. If you are validating user input, it has to be perfect. If you are parsing data files from a known source that generates its files in the same way every time, our last attempt is probably more than sufficient to parse the data without errors. You can find a better regex to match dates in the example section.

Use Negated Character Sets Instead of the Dot

I will explain this in depth when I present you the repeat operators star and plus, but the warning is important enough to mention it here as well. I will illustrate with an example.

Suppose you want to match a double-quoted string. Sounds easy. We can have any number of any character between the double quotes, so `".*"` seems to do the trick just fine. The dot matches any character, and the star allows the dot to be repeated any number of times, including zero. If you test this regex on `Put a "string" between double quotes`, it will match `"string"` just fine. Now go ahead and test it on `Houston, we have a problem with "string one" and "string two". Please respond.`

Ouch. The regex matches `"string one" and "string two"`. Definitely not what we intended. The reason for this is that the star is *greedy*.

In the date-matching example, we improved our regex by replacing the dot with a character class. Here, we will do the same. Our original definition of a double-quoted string was faulty. We do not

want any number of *any character* between the quotes. We want any number of characters that are not double quotes or newlines between the quotes. So the proper regex is `"[^"\r\n]*"`.

Start of String and End of String Anchors

Thus far, I have explained literal characters and character classes. In both cases, putting one in a regex will cause the regex engine to try to match a single character.

Anchors are a different breed. They do not match any character at all. Instead, they match a position before, after or between characters. They can be used to "anchor" the regex match at a certain position. The caret `^` matches the position before the first character in the string. Applying `^a` to `abc` matches `a`. `^b` will not match `abc` at all, because the `b` cannot be matched right after the start of the string, matched by `^`. See below for the inside view of the regex engine.

Similarly, `$` matches right after the last character in the string. `c$` matches `c` in `abc`, while `a$` does not match at all.

Useful Applications

When using regular expressions in a programming language to validate user input, using anchors is very important. If you use the code `if ($input =~ m/\d+/)` in a Perl script to see if the user entered an integer number, it will accept the input even if the user entered `qsdf4ghjk`, because `\d+` matches the 4. The correct regex to use is `^\d+$`. Because "start of string" must be matched before the match of `\d+`, and "end of string" must be matched right after it, the entire string must consist of digits for `^\d+$` to be able to match.

It is easy for the user to accidentally type in a space. When Perl reads from a line from a text file, the line break will also be stored in the variable. So before validating input, it is good practice to trim leading and trailing whitespace. `^\s+` matches leading whitespace and `\s+$` matches trailing whitespace. In Perl, you could use `$input =~ s/^\s+|\s+$//g`. Handy use of alternation and /g allows us to do this in a single line of code.

Using ^ and $ as Start of Line and End of Line Anchors

If you have a string consisting of multiple lines, like `first line\nsecond line` (where \n indicates a line break), it is often desirable to work with lines, rather than the entire string. Therefore, all the regex engines discussed in this tutorial have the option to expand the meaning of both anchors. `^` can then match at the start of the string (before the `f` in the above string), as well as after each line break (between `\n` and `s`). Likewise, `$` will still match at the end of the string (after the last `e`), and also before every line break (between `e` and `\n`).

In text editors like EditPad Pro or GNU Emacs, and regex tools like PowerGREP, the caret and dollar always match at the start and end of each line. This makes sense because those applications are designed to work with entire files, rather than short strings.

In every programming language and regex library I know, you have to explicitly activate this extended functionality. It is traditionally called "multi-line mode". In Perl, you do this by adding an m after the regex code, like this: `m/^regex$/m;`. In .NET, the anchors match before and after newlines when you specify `RegexOptions.Multiline`, such as in `Regex.Match("string", "regex", RegexOptions.Multiline)`.

Permanent Start of String and End of String Anchors

`\A` only ever matches at the start of the string. Likewise, `\Z` only ever matches at the end of the string. These two tokens never match at line breaks. This is true in all regex flavors discussed in this tutorial, even when you turn on "multiline mode". In EditPad Pro and PowerGREP, where the caret and dollar always match at the start and end of lines, `\A` and `\Z` only match at the start and the end of the entire file.

Zero-Length Matches

We saw that the anchors match at a position, rather than matching a character. This means that when a regex only consists of one or more anchors, it can result in a zero-length match. Depending on the situation, this can be very useful or undesirable. Using `^\d*$` to test if the user entered a number (notice the use of the star instead of the plus), would cause the script to accept an empty string as a valid input. See below.

However, matching only a position can be very useful. In email, for example, it is common to prepend a "greater than" symbol and a space to each line of the quoted message. In VB.NET, we can easily do this with `Dim Quoted as String = Regex.Replace(Original, "^", "> ", RegexOptions.Multiline)`. We are using multi-line mode, so the regex `^` matches at the start of the quoted message, and after each newline. The Regex.Replace method will remove the regex match from the string, and insert the replacement string (greater than symbol and a space). Since the match does not include any characters, nothing is deleted. However, the match does include a starting position, and the replacement string is inserted there, just like we want it.

Strings Ending with a Line Break

Even though `\Z` and `$` only match at the end of the string (when the option for the caret and dollar to match at embedded line breaks is off), there is one exception. If the string ends with a line break, then `\Z` and `$` will match at the position before that line break, rather than at the very end of the string. This "enhancement" was introduced by Perl, and is copied by many regex flavors, including Java, .NET and PCRE. In Perl, when reading a line from a file, the resulting string will end with a line break. Reading a line from a file with the text "joe" results in the string `joe\n`. When applied to this string, both `^[a-z]+$` and `\A[a-z]+\Z` will match `joe`.

If you only want a match at the absolute very end of the string, use `\z` (lower case z instead of upper case Z). `\A[a-z]+\z` does not match `joe\n`. `\z` matches after the line break, which is not matched by the character class.

Looking Inside the Regex Engine

Let's see what happens when we try to match `^4$` to `749\n486\n4` (where \n represents a newline character) in multi-line mode. As usual, the regex engine starts at the first character: `7`. The first token in the regular expression is `^`. Since this token is a zero-width token, the engine does not try to match it with the character, but rather with the position before the character that the regex engine has reached so far. `^` indeed matches the position before `7`. The engine then advances to the next regex token: `4`. Since the previous token was zero-width, the regex engine does *not* advance to the next character in the string. It remains at `7`. `4` is a literal character, which does not match `7`. There are no other permutations of the regex, so the engine starts again with the first regex token, at the next character: `4`. This time, `^` cannot match at the position before the 4. This position is preceded by a character, and that character is not a newline. The engine continues at `9`, and fails again. The next attempt, at `\n`, also fails. Again, the position before `\n` is preceded by a character, `9`, and that character is not a newline.

Then, the regex engine arrives at the second `4` in the string. The `^` can match at the position before the `4`, because it is preceded by a newline character. Again, the regex engine advances to the next regex token, `4`, but does not advance the character position in the string. `4` matches `4`, and the engine advances both the regex token and the string character. Now the engine attempts to match `$` at the position before (indeed: before) the `8`. The dollar cannot match here, because this position is preceded by a character, and that character is not a newline.

Yet again, the engine must try to match the first token again. Previously, it was successfully matched at the second `4`, so the engine continues at the next character, `8`, where the caret does not match. Same at the six and the newline.

Finally, the regex engine tries to match the first token at the third `4` in the string. With success. After that, the engine successfully matches `4` with `4`. The current regex token is advanced to `$`, and the current character is advanced to the very last position in the string: the void after the string. No regex token that needs a character to match can match here. Not even a negated character class. However, we are trying to match a dollar sign, and the mighty dollar is a strange beast. It is zero-width, so it will try to match the position before the current character. It does not matter that this "character" is the void after the string. In fact, the dollar will check the current character. It must be either a newline, or the void after the string, for `$` to match the position before the current character. Since that is the case after the example, the dollar matches successfully.

Since `$` was the last token in the regex, the engine has found a successful match: the last `4` in the string.

Another Inside Look

Earlier I mentioned that `^\d*$` would successfully match an empty string. Let's see why.

There is only one "character" position in an empty string: the void after the string. The first token in the regex is `^`. It matches the position before the void after the string, because it is preceded by the void before the string. The next token is `\d*`. As we will see later, one of the star's effects is that it makes the `\d`, in this case, optional. The engine will try to match `\d` with the void after the string. That fails, but the star turns the failure of the `\d` into a zero-width success. The engine will proceed with the next regex token, without advancing the position in the string. So the engine arrives at `$`, and the void after the string. We already saw that those match. At this point, the entire regex has matched the empty string, and the engine reports success.

Caution for Programmers

A regular expression such as `$` all by itself can indeed match after the string. If you would query the engine for the character position, it would return the length of the string if string indices are zero-based, or the length+1 if string indices are one-based in your programming language. If you would query the engine for the length of the match, it would return zero.

What you have to watch out for is that String[Regex.MatchPosition] may cause an access violation or segmentation fault, because MatchPosition can point to the void after the string. This can also happen with `^` and `^$` if the last character in the string is a newline.

Word Boundaries

The metacharacter `\b` is an anchor like the caret and the dollar sign. It matches at a position that is called a "word boundary". This match is zero-length.

There are four different positions that qualify as word boundaries:

- Before the first character in the string, if the first character is a word character.
- After the last character in the string, if the last character is a word character.
- Between a word character and a non-word character following right after the word character.
- Between a non-word character and a word character following right after the non-word character.

Simply put: `\b` allows you to perform a "whole words only" search using a regular expression in the form of `\bword\b`. A "word character" is a character that can be used to form words. All characters that are not "word characters" are "non-word characters". The exact list of characters is different for each regex flavor, but all word characters are always matched by the short-hand character class `\w`. All non-word characters are always matched by `\W`.

In Perl and the other regex flavors discussed in this tutorial, there is only one metacharacter that matches both before a word and after a word. This is because any position between characters can never be both at the start and at the end of a word. Using only one operator makes things easier for you.

Note that `\w` usually also matches digits. So `\b4\b` can be used to match a 4 that is not part of a larger number. This regex will not match `44 sheets of a4`. So saying "`\b` matches before and after an alphanumeric sequence" is more exact than saying "before and after a word".

Negated Word Boundary

`\B` is the negated version of `\b`. `\B` matches at every position where `\b` does not. Effectively, `\B` matches at any position between two word characters as well as at any position between two non-word characters.

Looking Inside the Regex Engine

Let's see what happens when we apply the regex `\bis\b` to the string `This island is beautiful`. The engine starts with the first token `\b` at the first character `T`. Since this token is zero-length, the position before the character is inspected. `\b` matches here, because the T is a word character and the character before it is the void before the start of the string. The engine continues with the next token: the literal `i`. The engine does not advance to the next character in the string, because the previous regex token was zero-width. `i` does not match `T`, so the engine retries the first token at the next character position.

`\b` cannot match at the position between the `T` and the `h`. It cannot match between the `h` and the `i` either, and neither between the `i` and the `s`.

The next character in the string is a space. `\b` matches here because the space is not a word character, and the preceding character is. Again, the engine continues with the `i` which does not match with the space.

Advancing a character and restarting with the first regex token, `\b` matches between the space and the second `i` in the string. Continuing, the regex engine finds that `i` matches `i` and `s` matches `s`. Now, the engine tries to match the second `\b` at the position before the `l`. This fails because this position is between two word characters. The engine reverts to the start of the regex and advances one character to the `s` in `island`. Again, the `\b` fails to match and continues to do so until the second space is reached. It matches there, but matching the `i` fails.

But `\b` matches at the position before the third `i` in the string. The engine continues, and finds that `i` matches `i` and `s` matches `s`. The last token in the regex, `\b`, also matches at the position before the second space in the string because the space is not a word character, and the character before it is.

The engine has successfully matched the word `is` in our string, skipping the two earlier occurrences of the characters i and s. If we had used the regular expression `is`, it would have matched the `is` in `This`.

Alternation with The Vertical Bar or Pipe Symbol

I already explained how you can use character classes to match a single character out of several possible characters. Alternation is similar. You can use alternation to match a single regular expression out of several possible regular expressions.

If you want to search for the literal text `cat` or `dog`, separate both options with a vertical bar or pipe symbol: `cat|dog`. If you want more options, simply expand the list: `cat|dog|mouse|fish`.

The alternation operator has the lowest precedence of all regex operators. That is, it tells the regex engine to match either everything to the left of the vertical bar, or everything to the right of the vertical bar. If you want to limit the reach of the alternation, you will need to use round brackets for grouping. If we want to improve the first example to match whole words only, we would need to use `\b(cat|dog)\b`. This tells the regex engine to find a word boundary, then either "cat" or "dog", and then another word boundary. If we had omitted the round brackets, the regex engine would have searched for "a word boundary followed by cat", or, "dog followed by a word boundary.

Remember That The Regex Engine Is Eager

I already explained that the regex engine is eager. It will stop searching as soon as it finds a valid match. The consequence is that in certain situations, the order of the alternatives matters. Suppose you want to use a regex to match a list of function names in a programming language: Get, GetValue, Set or SetValue. The obvious solution is `Get|GetValue|Set|SetValue`. Let's see how this works out when the string is `SetValue`.

The regex engine starts at the first token in the regex, `G`, and at the first character in the string, `S`. The match fails. However, the regex engine studied the entire regular expression before starting. So it knows that this regular expression uses alternation, and that the entire regex has not failed yet. So it continues with the second option, being the second `G` in the regex. The match fails again. The next token is the first `S` in the regex. The match succeeds, and the engine continues with the next character in the string, as well as the next token in the regex. The next token in the regex is the `e` after the `S` that just successfully matched. `e` matches `e`. The next token, `t` matches `t`.

At this point, the third option in the alternation has been successfully matched. Because the regex engine is eager, it considers the entire alternation to have been successfully matched as soon as one of the options has. In this example, there are no other tokens in the regex outside the alternation, so the entire regex has successfully matched `Set` in `SetValue`.

Contrary to what we intended, the regex did not match the entire string. There are several solutions. One option is to take into account that the regex engine is eager, and change the order of the options. If we use `GetValue|Get|SetValue|Set`, `SetValue` will be attempted before `Set`, and the engine will match the entire string. We could also combine the four options into two and use the question mark to make part of them optional: `Get(Value)?|Set(Value)?`. Because the question mark is greedy, `SetValue` will be attempted before `Set`.

The best option is probably to express the fact that we only want to match complete words. We do not want to match Set or SetValue if the string is `SetValueFunction`. So the solution is

`\b(Get|GetValue|Set|SetValue)\b` or `\b(Get(Value)?|Set(Value)?)\b`. Since all options have the same end, we can optimize this further to `\b(Get|Set)(Value)?\b`.

Optional Items

The question mark makes the preceding token in the regular expression optional. E.g.: `colou?r` matches both `colour` and `color`.

You can make several tokens optional by grouping them together using round brackets, and placing the question mark after the closing bracket. E.g.: `Nov(ember)?` will match `Nov` and `November`.

You can write a regular expression that matches many alternatives by including more than one question mark. `Feb(ruary)? 23(rd)?` matches `February 23rd`, `February 23`, `Feb 23rd` and `Feb 23`.

Important Regex Concept: Greediness

With the question mark, I have introduced the first metacharacter that is *greedy*. The question mark gives the regex engine two choices: try to match the part the question mark applies to, or do not try to match it. The engine will always try to match that part. Only if this causes the entire regular expression to fail, will the engine try ignoring the part the question mark applies to.

The effect is that if you apply the regex `Feb 23(rd)?` to the string `Today is Feb 23rd, 2003`, the match will always be `Feb 23rd` and not `Feb 23`. You can make the question mark *lazy* (i.e. turn off the greediness) by putting a second question mark after the first.

I will say a lot more about greediness when discussing the other repetition operators.

Looking Inside The Regex Engine

Let's apply the regular expression `colou?r` to the string `The colonel likes the color green`.

The first token in the regex is the literal `c`. The first position where it matches successfully is the `c` in `colonel`. The engine continues, and finds that `o` matches `o`, `l` matches `l` and another `o` matches `o`. Then the engine checks whether `u` matches `n`. This fails. However, the question mark tells the regex engine that failing to match `u` is acceptable. Therefore, the engine will skip ahead to the next regex token: `r`. But this fails to match `n` as well. Now, the engine can only conclude that the entire regular expression cannot be matched starting at the `c` in `colonel`. Therefore, the engine starts again trying to match `c` to the first o in `colonel`.

After a series of failures, `c` will match with the `c` in `color`, and `o`, `l` and `o` match the following characters. Now the engine checks whether `u` matches `r`. This fails. Again: no problem. The question mark allows the engine to continue with `r`. This matches `r` and the engine reports that the regex successfully matched `color` in our string.

Repetition with Star and Plus

I already introduced one repetition operator or quantifier: the question mark. It tells the engine to attempt match the preceding token zero times or once, in effect making it optional.

The asterisk or star tells the engine to attempt to match the preceding token zero or more times. The plus tells the engine to attempt to match the preceding token once or more. `<[A-Za-z][A-Za-z0-9]*>` matches an HTML tag without any attributes. The sharp brackets are literals. The first character class matches a letter. The second character class matches a letter or digit. The star repeats the second character class. Because we used the star, it's OK if the second character class matches nothing. So our regex will match a tag like `<B>`. When matching `<HTML>`, the first character class will match `H`. The star will cause the second character class to be repeated three times, matching `T`, `M` and `L` with each step.

I could also have used `<[A-Za-z0-9]+>`. I did not, because this regex would match `<1>`, which is not a valid HTML tag. But this regex may be sufficient if you know the string you are searching through does not contain any such invalid tags.

Limiting Repetition

Modern regex flavors, like those discussed in this tutorial, have an additional repetition operator that allows you to specify how many times a token can be repeated. The syntax is `{`*`min`*`,`*`max`*`}`, where *min* is a positive integer number indicating the minimum number of matches, and *max* is an integer equal to or greater than *min* indicating the maximum number of matches. If the comma is present but *max* is omitted, the maximum number of matches is infinite. So `{0,}` is the same as `*`, and `{1,}` is the same as `+`. Omitting both the comma and *max* tells the engine to repeat the token exactly *min* times.

You could use `\b[1-9][0-9]{3}\b` to match a number between 1000 and 9999. `\b[1-9][0-9]{2,4}\b` matches a number between 100 and 99999. Notice the use of the word boundaries.

Watch Out for The Greediness!

Suppose you want to use a regex to match an HTML tag. You know that the input will be a valid HTML file, so the regular expression does not need to exclude any invalid use of sharp brackets. If it sits between sharp brackets, it is an HTML tag.

Most people new to regular expressions will attempt to use `<.+>`. They will be surprised when they test it on a string like `This is a <EM>first</EM> test`. You might expect the regex to match `<EM>` and when continuing after that match, `</EM>`.

But it does not. The regex will match `<EM>first</EM>`. Obviously not what we wanted. The reason is that the plus is *greedy*. That is, the plus causes the regex engine to repeat the preceding token as often as possible. Only if that causes the entire regex to fail, will the regex engine *backtrack*. That is, it will go back to the plus, make it give up the last iteration, and proceed with the remainder of the

regex. Let's take a look inside the regex engine to see in detail how this works and why this causes our regex to fail. After that, I will present you with two possible solutions.

Like the plus, the star and the repetition using curly braces are greedy.

Looking Inside The Regex Engine

The first token in the regex is `<`. This is a literal. As we already know, the first place where it will match is the first `<` in the string. The next token is the dot, which matches any character except newlines. The dot is repeated by the plus. The plus is *greedy*. Therefore, the engine will repeat the dot as many times as it can. The dot matches `E`, so the regex continues to try to match the dot with the next character. `M` is matched, and the dot is repeated once more. The next character is the `>`. You should see the problem by now. The dot matches the `>`, and the engine continues repeating the dot. The dot will match all remaining characters in the string. The dot fails when the engine has reached the void after the end of the string. Only at this point does the regex engine continue with the next token: `>`.

So far, `<.+` has matched `<EM>first</EM> test` and the engine has arrived at the end of the string. `>` cannot match here. The engine remembers that the plus has repeated the dot more often than is required. (Remember that the plus *requires* the dot to match only once.) Rather than admitting failure, the engine will *backtrack*. It will reduce the repetition of the plus by one, and then continue trying the remainder of the regex.

So the match of `.+` is reduced to `EM>first</EM> tes`. The next token in the regex is still `>`. But now the next character in the string is the last `t`. Again, these cannot match, causing the engine to backtrack further. The total match so far is reduced to `<EM>first</EM> te`. But `>` still cannot match. So the engine continues backtracking until the match of `.+` is reduced to `EM>first</EM`. Now, `>` can match the next character in the string. The last token in the regex has been matched. The engine reports that `<EM>first</EM>` has been successfully matched.

Remember that the regex engine is *eager* to return a match. It will not continue backtracking further to see if there is another possible match. It will report the first valid match it finds. Because of greediness, this is the leftmost longest match.

Laziness Instead of Greediness

The quick fix to this problem is to make the plus lazy instead of greedy. You can do that by putting a question mark behind the plus in the regex. You can do the same with the star, the curly braces and the question mark itself. So our example becomes `<.+?>`. Let's have another look inside the regex engine.

Again, `<` matches the first `<` in the string. The next token is the dot, this time repeated by a lazy plus. This tells the regex engine to repeat the dot as few times as possible. The minimum is one. So the engine matches the dot with `E`. The requirement has been met, and the engine continues with `>` and `M`. This fails. Again, the engine will *backtrack*. But this time, the backtracking will force the

lazy plus to expand rather than reduce its reach. So the match of `.+` is expanded to `EM`, and the engine tries again to continue with `>`. Now, `>` is matched successfully. The last token in the regex has been matched. The engine reports that `<EM>` has been successfully matched. That's more like it.

An Alternative to Laziness

In this case, there is a better option than making the plus lazy. We can use a greedy plus and a negated character class: `<[^>]+>`. The reason why this is better is because of the backtracking. When using the lazy plus, the engine has to backtrack for each character in the HTML tag that it is trying to match. When using the negated character class, no backtracking occurs at all when the string contains valid HTML code. Backtracking slows down the regex engine. You will not notice the difference when doing a single search in a text editor. But you will save plenty of CPU cycles when using such a regex is used repeatedly in a tight loop in a script that you are writing, or perhaps in a custom syntax coloring scheme for EditPad Pro.

Finally, remember that this tutorial only talks about regex-directed engines. Text-directed engines do not backtrack. They do not get the speed penalty, but they also do not support lazy repetition operators.

Use Round Brackets for Grouping

By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together. This allows you to apply a regex operator, e.g. a repetition operator, to the entire group. I have already used round brackets for this purpose in previous topics throughout this tutorial.

Note that only round brackets can be used for grouping. Square brackets define a character class, and curly braces are used by a special repetition operator.

Round Brackets Create a Backreference

Besides grouping part of a regular expression together, round brackets also create a "backreference". A backreference stores the part of the string matched by the part of the regular expression inside the parentheses.

That is, unless you use non-capturing parentheses. Remembering part of the regex match in a backreference, slows down the regex engine because it has more work to do. If you do not use the backreference, you can speed things up by using non-capturing parentheses, at the expense of making your regular expression slightly harder to read.

The regex `Set(Value)?` matches `Set` or `SetValue`. In the first case, the first backreference will be empty, because it did not match anything. In the second case, the first backreference will contain `Value`.

If you do not use the backreference, you can optimize this regular expression into `Set(?:Value)?`. The question mark and the colon after the opening round bracket are the special syntax that you can use to tell the regex engine that this pair of brackets should not create a backreference. Note the question mark after the opening bracket is unrelated to the question mark at the end of the regex. That question mark is the regex operator that makes the previous token optional. This operator cannot appear after an opening round bracket, because an opening bracket by itself is not a valid regex token. Therefore, there is no confusion between the question mark as an operator to make a token optional, and the question mark as a character to change the properties of a pair of round brackets. The colon indicates that the change we want to make is to turn off capturing the backreference.

How to Use Backreferences

Backreferences allow you to reuse part of the regex match. You can reuse it inside the regular expression (see below), or afterwards. What you can do with it afterwards, depends on the tool you are using. In EditPad Pro or PowerGREP, you can use the backreference in the replacement text during a search-and-replace operation by typing `\1` (backslash one) into the replacement text. If you searched for `EditPad (Lite|Pro)` and use `\1 version` as the replacement, the actual replacement will be `Lite version` in case `EditPad Lite` was matched, and `Pro version` in case `EditPad Pro` was matched.

EditPad Pro and PowerGREP have a unique feature that allows you to change the case of the backreference. `\U1` inserts the first backreference in uppercase, `\L1` in lowercase and `\F1` with the first character in uppercase and the remainder in lowercase. Finally, `\I1` inserts it with the first letter of each word capitalized, and the other letters in lowercase.

Regex libraries in programming languages also provide access to the backreference. In Perl, you can use the magic variables `$1`, `$2`, etc. to access the part of the string matched by the backreference. In .NET (dot net), you can use the `Match` object that is returned by the `Match` method of the `Regex` class. This object has a property called `Groups`, which is a collection of Group objects. To get the string matched by the third backreference in C#, you can use `MyMatch.Groups[3].Value`.

The .NET (dot net) Regex class also has a method `Replace` that can do a regex-based search-and-replace on a string. In the replacement text, you can use `$1`, `$2`, etc. to insert backreferences.

To figure out the number of a particular backreference, scan the regular expression from left to right and count the opening round brackets. The first bracket starts backreference number one, the second number two, etc. Non-capturing parentheses are not counted. This fact means that non-capturing parentheses have another benefit: you can insert them into a regular expression without changing the numbers assigned to the backreferences. This can be very useful when modifying a complex regular expression.

The Entire Regex Match As Backreference Zero

Certain tools make the entire regex match available as backreference zero. In EditPad Pro or PowerGREP, you can use the entire regex match in the replacement text during a search and replace operation by typing `\0` (backslash zero) into the replacement text. In Perl, the magic variable `$&` holds the entire regex match. Libraries like .NET (dot net) where backreferences are made available as an array or numbered list, the item with index zero holds the entire regex match. Using backreference zero is more efficient than putting an extra pair of round brackets around the entire regex, because that would force the engine to continuously keep an extra copy of the entire regex match.

Using Backreferences in The Regular Expression

Backreferences can not only be used after a match has been found, but also during the match. Suppose you want to match a pair of opening and closing HTML tags, and the text in between. By putting the opening tag into a backreference, we can reuse the name of the tag for the closing tag. Here's how: `<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>`. This regex contains only one pair of parentheses, which capture the string matched by `[A-Z][A-Z0-9]` into the first backreference. This backreference is reused with `\1` (backslash one). The `/` before it is simply the forward slash in the closing HTML tag that we are trying to match.

You can reuse the same backreference more than once. `([a-c])x\1x\1` will match `axaxa`, `bxbxb` and `cxcxc`. If a backreference was not used in a particular match attempt (such as in the first example where the question mark made the first backreference optional), it is simply empty. Using an empty backreference in the regex is perfectly fine. It will simply be replaced with nothingness.

A backreference cannot be used inside itself. `([abc]\1)` will not work. Depending on your regex flavor, it will either give an error message, or it will fail to match anything without an error message. Therefore, \0 cannot be used inside a regex, only in the replacement.

Looking Inside The Regex Engine

Let's see how the regex engine applies the above regex to the string `Testing <B><I>bold italic</I></B> text`. The first token in the regex is the literal `<`. The regex engine will traverse the string until it can match at the first `<` in the string. The next token is `[A-Z]`. The regex engine also takes note that it is now inside the first pair of capturing parentheses. `[A-Z]` matches `B`. The engine advances to `[A-Z0-9]` and `>`. This match fails. However, because of the star, that's perfectly fine. The position in the string remains at `>`. The position in the regex is advanced to `[^>]`.

This step crosses the closing bracket of the first pair of capturing parentheses. This prompts the regex engine to store what was matched inside them into the first backreference. In this case, `B` is stored.

After storing the backreference, the engine proceeds with the match attempt. `[^>]` does not match `>`. Again, because of another star, this is not a problem. The position in the string remains at `>`, and position in the regex is advanced to `>`. These obviously match. The next token is a dot, repeated by a lazy star. Because of the laziness, the regex engine will initially skip this token, taking note that it should backtrack in case the remainder of the regex fails.

The engine has now arrived at the second `<` in the regex, and the second `<` in the string. These match. The next token is `/`. This does not match `I`, and the engine is forced to backtrack to the dot. The dot matches the second `<` in the string. The star is still lazy, so the engine again takes note of the available backtracking position and advances to `<` and `I`. These do not match, so the engine again backtracks.

The backtracking continues until the dot has consumed `<I>bold italic`. At this point, `<` matches the third `<` in the string, and the next token is `/` which matches `/`. The next token is `\1`. Note that the token the backreference, and not `B`. The engine does not substitute the backreference in the regular expression. Every time the engine arrives at the backreference, it will read the value that was stored. This means that if the engine had backtracked beyond the first pair of capturing parentheses before arriving the second time at `\1`, the new value stored in the first backreference would be used. But this did not happen here, so `B` it is. This fails to match at `I`, so the engine backtracks again, and the dot consumes the third `<` in the string.

Backtracking continues again until the dot has consumed `<I>bold italic</I>`. At this point, `<` matches `<` and `/` matches `/`. The engine arrives again at `\1`. The backreference still holds `B`. `B` matches `B`. The last token in the regex, `>` matches `>`. A complete match has been found: `<B><I>bold italic</I></B>`.

Repetition and Backreferences

As I mentioned in the above inside look, the regex engine does not permanently substitute backreferences in the regular expression. It will use the last match saved into the backreference each time it needs to be used. If a new match is found by capturing parentheses, the previously saved match is overwritten. There is a clear difference between `([abc]+)` and `([abc])+`. Though both successfully match `cab`, the first regex will put `cab` into the first backreference, while the second regex will only store `b`. That is because in the second regex, the plus caused the pair of parentheses to repeat three times. The first time, `c` was stored. The second time `a` and the third time `b`. Each time, the previous value was overwritten, so `b` remains.

This also means that `([abc]+)=\1` will match `cab=cab`, and that `([abc])+=\1` will not. The reason is that when the engine arrives at `\1`, it holds `b` which fails to match `c`. Obvious when you look at a simple example like this one, but a common cause of difficulty with regular expressions nonetheless. When using backreferences, always double check that you are really capturing what you want.

Useful Example: Checking for Doubled Words

When editing text, doubled words such as "the the" easily creep in. Using the regex `\b(\w+)\s+\1\b` in your text editor, you can easily find them. To delete the second word, simply type in `\1` as the replacement text and click the Replace button.

Parentheses and Backreferences Cannot Be Used Inside Character Classes

Round brackets cannot be used inside character classes, at least not as metacharacters. When you put a round bracket in a character class, it is treated as a literal character. So the regex `[(a)b]` matches `a`, `b`, `(` and `)`.

Backreferences also cannot be used inside a character class. The \1 in regex like `(a)[\1b]` will be interpreted as an octal escape in most regex flavors. So this regex will match an a followed by either `\x01` or a `b`.

Named Capturing Groups

All modern regular expression engines support capturing groups, which are numbered from left to right, starting with one. The numbers can then be used in backreferences to match the same text again in the regular expression, or to use part of the regex match for further processing. In a complex regular expression with many capturing groups, the numbering can get a little confusing.

Named Capture with Python, PCRE and PHP

Python's regex module was the first to offer a solution: named capture. By assigning a name to a capturing group, you can easily reference it by name. `(?P<name>group)` captures the match of `group` into the backreference "name". You can reference the contents of the group with the numbered backreference `\1` or the named backreference `(?P=name)`.

The open source PCRE library has followed Python's example, and offers named capture using the same syntax. The PHP preg functions offer the same functionality, since they are based on PCRE.

Python's `sub()` function allows you to reference a named group as `\1` or `\g<name>`. This does *not* work in PHP. In PHP, you can use double-quoted string interpolation with the `$regs` parameter you passed to `pcre_match()`: `$regs['name']`.

Named Capture with .NET's System.Text.RegularExpressions

The regular expression classes of the .NET framework also support named capture. Unfortunately, the Microsoft developers decided to invent their own syntax, rather than follow the one pioneered by Python. Currently, no other regex flavor supports Microsoft's version of named capture. RegexBuddy supports both Python's and Microsoft's style, and will convert one flavor of named capture into the

other when generating source code snippets for Python, PHP/preg, PHP, or one of the .NET languages.

Here is an example with two capturing groups in .NET style: `(?<first>group)(?'second'group)`. As you can see, .NET offers two syntaxes to create a capturing group: one using sharp brackets, and the other using single quotes. The first syntax is preferable in strings, where single quotes may need to be escaped. The second syntax is preferable in ASP code, where the sharp brackets are used for HTML tags. You can use the pointy bracket flavor and the quoted flavors interchangeably.

To reference a capturing group inside the regex, use `\k<name>` or `\k'name'`. Again, you can use the two syntactic variations interchangeably.

When doing a search-and-replace, you can reference the named group with the familiar dollar sign syntax: `${name}`. Simply use a name instead of a number between the curly braces.

Names and Numbers for Capturing Groups

Here is where things get a bit ugly. Python and PCRE treat named capturing groups just like unnamed capturing groups, and number both kinds from left to right, starting with one. The regex `(a)(?P<x>b)(c)(?P<y>d)` matches `abcd` as expected. If you do a search-and-replace with this regex and the replacement `\1\2\3\4`, you will get `abcd`. All four groups were numbered from left to right, from one till four. Easy and logical.

Things are quite a bit more complicated with the .NET framework. The regex `(a)(?<x>b)(c)(?<y>d)` again matches `abcd`. However, if you do a search-and-replace with `$1$2$3$4` as the replacement, you will get `acbd`. Probably not what you expected.

The .NET framework *does* number named capturing groups from left to right, but numbers them *after* all the unnamed groups have been numbered. So the unnamed groups `(a)` and `(c)` get numbered first, from left to right, starting at one. Then the named groups `(?<x>b)` and `(?<y>d)` get their numbers, continuing from the unnamed groups, in this case: three.

To make things simple, when using .NET's regex support, just assume that named groups do not get numbered at all, and reference them by name exclusively. To keep things compatible across regex flavors, I strongly recommend that you do not mix named and unnamed capturing groups at all. Either give a group a name, or make it non-capturing as in `(?:nocapture)`. Non-capturing groups are more efficient, since the regex engine does not need to keep track of their matches.

Regex Matching Modes

All regular expression engines discussed in this tutorial support the following three matching modes:

- `/i` makes the regex match case insensitive.
- `/s` enables "single-line mode". In this mode, the dot matches newlines.
- `/m` enables "multi-line mode". In this mode, the caret and dollar match before and after newlines in the subject string.

Many regex flavors have additional modes or options that have single letter equivalents, but these differ widely.

Most tools that support regular expressions have checkboxes or similar controls that you can use to turn these modes on or off. Most programming languages allow you to pass option flags when constructing the regex object. E.g. in Perl, `m/regex/i` turns on case insensitivity, while `Pattern.compile("regex", Pattern.CASE_INSENSITIVE)` does the same in Java.

Specifying Modes Inside The Regular Expression

Sometimes, the tool or language does not provide the ability to specify matching options. E.g. the handy `String.matches()` method in Java does not take a parameter for matching options like `Pattern.compile()` does.

In that situation, you can add a mode modifier to the start of the regex. E.g. `(?i)` turns on case insensitivity, while `(?ism)` turns on all three options.

Turning Modes On and Off for Only Part of The Regular Expression

Modern regex flavors allow you to apply modifiers to only part of the regular expression. If you insert the modifier `(?ism)` in the middle of the regex, the modifier only applies to the part of the regex to the right of the modifier. You can turn off a mode by preceding it with a minus sign. E.g. `(?i-sm)` turns on case insensitivity, turns off single-line mode, and turns on multi-line mode. To turn off several modes, precede each of their letters with a minus sign.

Not all regex flavors support this. The latest versions of all tools and languages discussed on this website do. Older regex flavors usually apply the option to the entire regular expression, no matter where you placed it.

You can quickly test this. The regex `(?i)te(?-i)st` should match `test` and `TEst`, but not `teST` or `TEST`.

Modifier Spans

Instead of using two modifiers, one to turn an option on, and one to turn it off, you use a modifier span. `(?i)ignorecase(?-i)casesensitive(?i)ignorecase` is equivalent to `(?i)ignorecase(?-i:casesensitive)ignorecase`. You have probably noticed the resemblance between the modifier span and the non-capturing group `(?:group)`. Technically, the non-capturing group is a modifier span that does not change any modifiers. It is obvious that the modifier span does not create a backreference.

Atomic Grouping and Possessive Quantifiers

When discussing the repetition operators or quantifiers, I explained the difference between greedy and lazy repetition. Greediness and laziness determine the order in which the regex engine tries the possible permutations of the regex pattern. A greedy quantifier will first try to repeat the token as many times as possible, and gradually give up matches as the engine backtracks to find an overall match. A lazy quantifier will first repeat the token as few times as required, and gradually expand the match as the engine backtracks through the regex to find an overall match.

Because greediness and laziness change the order in which permutations are tried, they can change the overall regex match. However, they do not change the fact that the regex engine will backtrack to try all possible permutations of the regular expression in case no match can be found. First, let's see why backtracking can lead to problems.

Catastrophic Backtracking

Recently I got a complaint from a customer that EditPad Pro hung (i.e. it stopped responding) when trying to find lines in a comma-delimited text file where the 12th item on a line started with a `P`. The customer was using the regexp `^(.*?,){11}P`.

At first sight, this regex looks like it should do the job just fine. The lazy dot and comma match a single comma-delimited field, and the {11} skips the first 11 fields. Finally, the P checks if the 12th field indeed starts with P. In fact, this is exactly what will happen when the 12th field indeed starts with a P.

The problem rears its ugly head when the 12th field does not start with a P. Let's say the string is `1,2,3,4,5,6,7,8,9,10,11,12,13`. At that point, the regex engine will backtrack. It will backtrack to the point where `^(.*?,){11}` had consumed `1,2,3,4,5,6,7,8,9,10,11`, giving up the last match of the comma. The next token is again the dot. The dot matches a comma. *The dot matches the comma!* However, the comma does not match the `1` in the 12th field, so the dot continues until the 11th iteration of `.*?,` has consumed `11,12.`. You can already see the root of the problem: the part of the regex (the dot) matching the contents of the field also matches the delimiter (the comma). Because of the double repetition (star inside {11}), this leads to a catastrophic amount of backtracking.

The regex engine now checks whether the 13th field starts with a P. It does not. Since there is no comma after the 13th field, the regex engine can no longer match the 11th iteration of `.*?,`. But it does not give up there. It backtracks to the 10th iteration, expanding the match of the 10th iteration to `10,11,`. Since there is still no P, the 10th iteration is expanded to `10,11,12,`. Reaching the end of the string again, the same story starts with the 9th iteration, subsequently expanding it to `9,10,`, `9,10,11,`, `9,10,11,12,`. But between each expansion, there are more possiblities to be tried. When the 9th iteration consumes `9,10,`, the 10th could match just `11,` as well as `11,12,`. Continuously failing, the engine backtracks to the 8th iteration, again trying all possible combinations for the 9th, 10th, and 11th iterations.

You get the idea: the possible number of combinations that the regex engine will try for each line where the 12th field does not start with a P is huge. This causes software like EditPad Pro to stop

responding, or even crash as the regex engine runs out of memory trying to remember all backtracking positions.

Preventing Catastrophic Backtracking

The solution is simple. When nesting repetition operators, make absolutely sure that there is only one way to match the same match. If repeating the inner loop 4 times and the outer loop 7 times results in the same overall match as repeating the inner loop 6 times and the outer loop 2 times, you can be sure that the regex engine will try all those combinations.

In our example, the solution is to be more exact about what we want to match. We want to match 11 comma-delimited fields. The fields must not contain comma's. So the regex becomes: `^([^,\r\n]*,){11}P`. If the P cannot be found, the engine will still backtrack. But it will backtrack only 11 times, and each time the `[^,\r\n]` is not able to expand beyond the comma, forcing the regex engine to the previous one of the 11 iterations immediately, without trying further options.

Atomic Grouping and Possessive Quantifiers

Recent regex flavors have introduced two additional solutions to this problem: atomic grouping and possessive quantifiers. Their purpose is to prevent backtracking, allowing the regex engine to fail faster.

In the above example, we could easily reduce the amount of backtracking to a very low level by better specifying what we wanted. But that is not always possible in such a straightforward manner. In that case, you should use atomic grouping to prevent the regex engine from backtracking.

Using atomic grouping, the above regex becomes `^(?>(.*?,){11})P`. Everything between `(?>)` is treated as one single token by the regex engine, once the regex engine leaves the group. Because the entire group is one token, no backtracking can take place once the regex engine has found a match for the group. If backtracking is required, the engine has to backtrack to the regex token before the group (the caret in our example). If there is no token before the group, the regex must retry the entire regex at the next position in the string.

Possessive quantifiers are a limited form of atomic grouping with a cleaner notation. To make a quantifier possessive, place a plus after it. `x++` is the same as `(?>x+)`. Similarly, you can use `x*+`, `x?+` and `x{m,n}+`. Note that you cannot make a lazy quantifier possessive. It would match the minimum number of matches and never expand the match because backtracking is not allowed.

Tool and Language Support for Atomic Grouping and Possessive Quantifiers

Atomic grouping is a recent addition to the regex scene, and only supported by the latest versions of most regex flavors. Perl supports it starting with version 5.6. The Java supports it starting with JDK

version 1.4.2, though the JDK documentation uses the term "independent group" rather than "atomic group". All versions of .NET support atomic grouping, as do recent versions of PCRE and PHP's pgreg functions. Python does not support atomic grouping.

At this time, possessive quantifiers are only supported by the Java JDK 1.4.0 and later, and PCRE version 4 and later.

The latest versions of EditPad Pro and PowerGREP support both atomic grouping and possessive quantifiers, as do all versions of RegexBuddy.

Atomic Grouping Inside The Regex Engine

Let's see how `^(?>(.*?,){11})P` is applied to `1,2,3,4,5,6,7,8,9,10,11,12,13`. The caret matches at the start of the string and the engine enters the atomic group. The star is lazy, so the dot is initially skipped. But the comma does not match `1`, so the engine backtracks to the dot. That's right: backtracking is allowed here. The star is not possessive, and is not immediately enclosed by an atomic group. That is, the regex engine did not cross the closing round bracket of the atomic group. The dot matches `1`, and the comma matches too. `{11}` causes further repetition until the atomic group has matched `1,2,3,4,5,6,7,8,9,10,11,`.

Now, the engine leaves the atomic group. Because the group is atomic, all backtracking information is discarded and the group is now considered a single token. The engine now tries to match `P` to the `1` in the 12th field. This fails.

So far, everything happened just like in the original, troublesome regular expression. Now comes the difference. `P` failed to match, so the engine backtracks. The previous token is an atomic group, so the group's entire match is discarded and the engine backtracks further to the caret. The engine now tries to match the caret at the next position in the string, which fails. The engine walks through the string until the end, and declares failure. Failure is declared after 30 attempts to match the caret, and just one attempt to match the atomic group, rather than after 30 attempts to match the caret and a huge number of attempts to try all combinations of both quantifiers in the regex.

That is what atomic grouping and possessive quantifiers are for: efficiency by disallowing backtracking. The most efficient regex for our problem at hand would be `^(?>((?>[^,\r\n]*),){11})P`, since possessive, greedy repetition of the star is faster than a backtracking lazy dot. If possessive quantifiers are available, you can reduce clutter by writing `^(?>([^,\r\n]*+,){11})P`.

When To Use Atomic Grouping or Possessive Quantifiers

Atomic grouping and possessive quantifiers speed up failure by eliminating backtracking. They do not speed up success, only failure. When nesting quantifiers like in the above example, you really should use atomic grouping and/or possessive quantifiers whenever possible. While `x[^x]*+x` and `x(?>[^x]*)x` fail faster than `x[^x]*x`, the increase in speed is minimal. If the final x in the regex cannot be matched, the regex engine backtracks once for each character matched by the star. With

simple repetition, the amount of time wasted with pointless backtracking increases in a linear fashion to the length of the string. With combined repetition, the amount of time wasted increases exponentially and will very quickly exhaust the capabilities of your computer. Still, if you are smart about combined repetition, you often can avoid the problem without atomic grouping as in the example above.

If you are simply doing a search in a text editor, using simple repetition, you will not earn back the extra time to type in the characters for the atomic grouping. If the regex will be used in a tight loop in an application, or process huge amounts of data, then atomic grouping may make a difference.

Note that atomic grouping and possessive quantifiers can alter the outcome of the regular expression match. `\d+6` will match `123456` in `123456789`. `\d++6` will not match at all. `\d+` will match the entire string. With the former regex, the engine backtracks until the 6 can be matched. In the latter case, no backtracking is allowed, and the match fails. Again, the cause of this is that the token `\d` that is repeated can also match the delimiter `6`. Sometimes this is desirable, often it is not.

This shows again that understanding how the regex engine works on the inside will enable you to avoid many pitfalls and craft efficient regular expressions that match exactly what you want.

Lookahead and Lookbehind Zero-Width Assertions

Perl 5 introduced two very powerful constructs: "lookahead" and "lookbehind". Collectively, these are called "lookaround". They are also called "zero-width assertions". They are zero-width just like the start and end of line, and start and end of word anchors that I already explained. The difference is that lookarounds will actually match characters, but then give up the match and only return the result: match or no match. That is why they are called "assertions". They do not consume characters in the string, but only assert whether a match is possible or not.

Lookarounds allow you to create regular expressions that are impossible to create without them, or that would get very longwinded without them. All regex flavors discussed on this web site support lookaround. The exception is JavaScript, which supports lookahead but not lookbehind.

Positive and Negative Lookahead

Negative lookahead is indispensable if you want to match something not followed by something else. When explaining character classes, I already explained why you cannot use a negated character class to match a "q" not followed by a "u". Negative lookahead provides the solution: `q(?!u)`. The negative lookahead construct is the pair of round brackets, with the opening bracket followed by a question mark and an explanation point. Inside the lookahead, we have the trivial regex `u`.

Positive lookahead works just the same. `q(?=u)` matches a q that is followed by a u, without making the u part of the match. The positive lookahead construct is a pair of round brackets, with the opening bracket followed by a question mark and an equals sign.

You can use any regular expression inside the lookahead. (Note that this is not the case with lookbehind. I will explain why below.) Any valid regular expression can be used inside the lookahead. If it contains capturing parentheses, the backreferences will be saved. Note that the lookahead itself does not create a backreference. So it is not included in the count towards numbering the backreferences. If you want to store the match of the regex inside a backreference, you have to put capturing parentheses around the regex inside the lookahead, like this: `(?=(regex))`. The other way around will not work, because the lookahead will already have discarded the regex match by the time the backreference is to be saved.

Regex Engine Internals

First, let's see how the engine applies `q(?!u)` to the string `Iraq`. The first token in the regex is the literal `q`. As we already know, this will cause the engine to traverse the string until the `q` in the string is matched. The position in the string is now the void behind the string. The next token is the lookahead. The engine takes note that it is inside a lookahead construct now, and begins matching the regex inside the lookahead. So the next token is `u`. This does not match the void behind the string. The engine notes that the regex inside the lookahead failed. Because the lookahead is negative, this means that the lookahead has successfully matched at the current position. At this point, the entire regex has matched, and `q` is returned as the match.

Let's try applying the same regex to `quit`. `q` matches `q`. The next token is the `u` inside the lookahead. The next character is the `u`. These match. The engine advances to the next character: `i`. However, it is done with the regex inside the lookahead. The engine notes success, and discards the regex match. This causes the engine to step back in the string to `u`.

Because the lookahead is negative, the successful match inside it causes the lookahead to fail. Since there are no other permutations of this regex, the engine has to start again at the beginning. Since `q` cannot match anywhere else, the engine reports failure.

Let's take one more look inside, to make sure you understand the implications of the lookahead. Let's apply `q(?=u)i` to `quit`. I have made the lookahead positive, and put a token after it. Again, `q` matches `q` and `u` matches `u`. Again, the match from the lookahead must be discarded, so the engine steps back from `i` in the string to `u`. To lookahead was successful, so the engine continues with `i`. But `i` cannot match `u`. So this match attempt fails. All remaining attempts will fail as well, because there are no more q's in the string.

Positive and Negative Lookbehind

Lookbehind has the same effect, but works backwards. It tells the regex engine to temporarily step backwards in the string, to check if the text inside the lookbehind can be matched there. `(?<!a)b` matches a "b" that is not preceded by an "a", using negative lookbehind. It will not match `cab`, but will match the `b` (and only the `b`) in `bed` or `debt`. `(?<=a)b` (positive lookbehind) matches the `b` (and only the `b`) in `cab`, but does not match `bed` or `debt`.

The construct for positive lookbehind is `(?<=text)`: a pair of round brackets, with the opening bracket followed by a question mark, "less than" symbol and an equals sign. Negative lookbehind is written as `(?<!text)`, using an exclamation point instead of an equals sign.

More Regex Engine Internals

Let's apply `(?<=a)b` to `thingamabob`. The engine starts with the lookbehind and the first character in the string. In this case, the lookbehind tells the engine to step back one character, and see if an "a" can be matched there. The engine cannot step back one character because there are no characters before the `t`. So the lookbehind fails, and the engine starts again at the next character, the `h`. (Note that a negative lookbehind would have succeeded here.) Again, the engine temporarily steps back one character to check if an "a" can be found there. It finds a `t`, so the positive lookbehind fails again.

The lookbehind continues to fail until the regex reaches the `m` in the string. The engine again steps back one character, and notices that the `a` can be matched there. The positive lookbehind matches. Because it is zero-width, the current position in the string remains at the `m`. The next token is `b`, which cannot match here. The next character is the second `a` in the string. The engine steps back, and finds out that the `m` does not match `a`.

The next character is the first `b` in the string. The engine steps back and finds out that `a` satisfies the lookbehind. `b` matches `b`, and the entire regex has been matched successfully. It matches one character: the first `b` in the string.

Important Notes About Lookbehind

The good news is that you can use lookbehind anywhere in the regex, not only at the start. If you want to find a word not ending with an "s", you could use `\b\w+(?<!s)\b`. This is definitely not the same as `\b\w+[^s]\b`. When applied to `John's`, the former will match `John` and the latter `John'` (including the apostrophe). I will leave it up to you to figure out why. (Hint: `\b` matches between the apostrophe and the `s`). The latter will also not match single-letter words like "a" or "I". The correct regex without using lookbehind is `\b\w*[^s\W]\b` (star instead of plus, and \W in the character class). Personally, I find the lookbehind easier to understand. The last regex, which works correctly, has a double negation (the \W in the negated character class). Double negations tend to be confusing to humans. Not to regex engines, though.

The bad news is that you cannot use just any regex inside a lookbehind. The reason is that regular expressions do not work backwards. The string must be traversed from left to right. Therefore, the regular expression engine needs to be able to figure out how many steps to step back before checking the lookbehind.

Therefore, many regex flavors, including those used by Perl 5 and Python, only allow fixed-length strings. You can use any regex of which the length of the match can be predetermined. This means

you can use literal text and character classes. You cannot use repetition or optional items. You can use alternation, but only if all options in the alternation have the same length.

Some regex flavors support the above, plus alternation with strings of different lengths. But each string in the alternation must still be of fixed length, so only literals and character classes can be used. This includes PCRE, PHP, RegexBuddy, EditPad Pro and PowerGREP.

Finally, some more advanced flavors support the above, plus finite repetition. This means you can still not use the star or plus, but you can use the question mark and the curly braces with the max parameter specified. These regex flavors recognize the fact that finite repetition can be rewritten as an alternation of strings with different, but fixed lengths. The only regex flavor that I know of that currently supports this is Sun's regex package in the JDK 1.4.

Even with these limitations, lookbehind is a valuable addition to the regular expression syntax.

Technically, the .NET framework can apply regular expressions backwards, and will allow you to use any regex, including infinite repetition, inside lookbehind. However, the semantics of applying a regular expression backwards are currently not well-defined. Microsoft has promised to resolve this in version 2.0 of the .NET framework. Until that happens, I recommend you use only fixed-length strings, alternation and character classes inside lookbehind.

Finally, JavaScript does not support lookbehind at all.

Testing The Same Part of The String for More Than One Requirement

Lookaround, which I introduced in detail in the previous topic, is a very powerful concept. Unfortunately, it is often underused by people new to regular expressions, because lookaround is a bit confusing. The confusing part is that the lookaround is zero-width. So if you have a regex in which a lookahead is followed by another piece of regex, or a lookbehind is preceded by another piece of regex, then the regex will traverse part of the string twice.

To make this clear, I would like to give you another, a bit more practical example. Let's say we want to find a word that is six letters long and contains the three subsequent letters `cat`. Actually, we can match this without lookaround. We just specify all the options and hump them together using alternation: `cat\w{3}|\wcat\w{2}|\w{2}cat\w|\w{3}cat`. Easy enough. But this method gets unwieldy if you want to find any word between 6 and 12 letters long containing either "cat", "dog" or "mouse".

Lookaround to The Rescue

In this example, we basically have two requirements for a successful match. First, we want a word that is 6 letters long. Second, the word we found must contain the word "cat".

Matching a 6-letter word is easy with `\b\w{6}\b`. Matching a word containing "cat" is equally easy: `\b\w*cat\w*\b`.

Combining the two, we get: `(?=\b\w{6}\b)\b\w*cat\w*\b`. Easy! Here's how this works. At each character position in the string where the regex is attempted, the engine will first attempt the regex inside the positive lookahead. This sub-regex, and therefore the lookahead, matches only when the current character position in the string is at the start of a 6-letter word in the string. If not, the lookahead will fail, and the engine will continue trying the regex from the start at the next character position in the string.

The lookahead is zero-width. So when the regex inside the lookahead has found the 6-letter word, the current position in the string is still at the beginning of the 6-letter word. At this position will the regex engine attempt the remainder of the regex. Because we already know that a 6-letter word can be matched at the current position, we know that `\b` matches and that the first `\w*` will match 6 times. The engine will then backtrack, reducing the number of characters matched by `\w*`, until `cat` can be matched. If `cat` cannot be matched, the engine has no other choice but to restart at the beginning of the regex, at the next character position in the string. This is at the second letter in the 6-letter word we just found, where the lookahead will fail, causing the engine to advance character by character until the next 6-letter word.

If `cat` can be successfully matched, the second `\w*` will consume the remaining letters, if any, in the 6-letter word. After that, the last `\b` in the regex is guaranteed to match where the second `\b` inside the lookahead matched. Our double-requirement-regex has matched successfully.

Optimizing Our Solution

While the above regex works just fine, it is not the most optimal solution. This is not a problem if you are just doing a search in a text editor. But optimizing things is a good idea if this regex will be used repeatedly and/or on large chunks of data in an application you are developing.

You can discover these optimizations by yourself if you carefully examine the regex and follow how the regex engine applies it, as I did above. I said the third and last `\b` are guaranteed to match. Since it is zero-width, and therefore does not change the result returned by the regex engine, we can remove them, leaving: `(?=\b\w{6}\b)\w*cat\w*`. Though the last `\w*` is also guaranteed to match, we cannot remove it because it adds characters to the regex match. Remember that the lookahead discards its match, so it does not contribute to the match returned by the regex engine. If we omitted the `\w*`, the resulting match would be the start of a 6-letter word containing "cat", up to and including "cat", instead of the entire word.

But we can optimize the first `\w*`. As it stands, it will match 6 letters and then backtrack. But we know that in a successful match, there can never be more than 3 letters before "cat". So we can optimize this to `\w{0,3}`. Note that making the asterisk lazy would not have optimized this sufficiently. The lazy asterisk would find a successful match sooner, but if a 6-letter word does not contain "cat", it would still cause the regex engine to try matching "cat" at the last two letters, at the last single letter, and even at one character beyond the 6-letter word.

So we have `(?=\b\w{6}\b)\w{0,3}cat\w*`. One last, minor, optimization involves the first `\b`. Since it is zero-width itself, there's no need to put it inside the lookahead. So the final regex is: `\b(?=\w{6}\b)\w{0,3}cat\w*`.

A More Complex Problem

So, what would you use to find any word between 6 and 12 letters long containing either "cat", "dog" or "mouse"? Again we have two requirements, which we can easily combine using a lookahead: `\b(?=\w{6,12}\b)\w{0,9}(cat|dog|mouse)\w*`. Very easy, once you get the hang of it. This regex will also put "cat", "dog" or "mouse" into the first backreference.

Finding Matches Only Inside a Section of The String

Lookahead allows you to create regular expression patterns that are impossible to create without it. One example is matching a particular regex only inside specific sections of the string or file searched through. To keep things simple, I will use `wanted` as a substitute for the regular expression that we are trying to match inside the section, and `start` as the regex matching the start of the section, and `stop` as the regex matching the end of the section.

If "wanted" occurs only once inside the section, we can do without lookahead. `start.*?wanted.*?stop` would do the trick. However, this will not work if "wanted" occurs more than once inside a single section. The reason is that this regular expression consumes the entire section. The entire section is included in the regex match. When we apply the regex again to the same string or file, it will continue after `stop`, not after `wanted`. So we need a way to match `wanted` without matching the rest of the section. This is possible with lookahead, because lookahead is zero-width.

You may be tempted to use a combination of lookbehind and lookahead like in `(?<=start.*?)wanted(?=.*?stop)`. However, this will not work. The regex engine will refuse to compile this regular expression. Lookbehind must be of fixed length. Since we do not know in advance how many characters there will be between "start" and "wanted", we need to use `.*?`. The star is obviously not of fixed length.

So we have to resort to using lookahead only. The final regular expression will be in the form of `wanted(?=insidesection)`. First we match the string we want, and then we test if it is inside the proper section.

How do we know if we matched `wanted` inside a section? First, we must be able to match `stop` after matching `wanted`. If not, we found a match after a section rather than inside a section. Second, we must not be able to match `start` between matching `wanted` and matching `stop`. If we could, we found a match before a section rather than inside a section. Note that these two rules will only yield success if the string or file searched through is properly translated into sections. That is, each match of `start` must be followed exactly by one match of `stop`.

So inside the lookahead we need to look for a series of unspecified characters that do not match the start of a section anywhere in the series, and end with the section stop. In a regex, this is written as: `((?!start).)*?stop`. The dot and negative lookahead match any character that is not the first character of the start of a section. This, we repeat zero or more times with the star. I used a lazy star to make the regex more efficient. After this, we need to match the end of the section. Effectively, the lazy star will continue to repeat until the end of the section is reached. Because of the negative

lookahead inside the star, the star will also stop at the start of a section, at which point stop cannot be matched and thus the regex will fail.

The final regular expression becomes: `wanted(?=((?!start).)*?stop)`. Substitute `wanted`, `start` and `stop` with the regexes of your choice.

Example: Search and Replace within Header Tags

Using the above generic regular expression, you can easily build a regex to do a search and replace on HTML files, replacing a certain word with another, but only inside title tags. A title tag starts with `<H[1-6]` and ends with `</H[1-6]>`. I omitted the closing > in the start tag to allow for attributes. So the regex becomes `wanted(?=((?!\<H[1-6]).)*?</H[1-6]>)`.

You may have noticed that I escaped the < of the opening tag in the final regex. I did that because some regex flavors interpret `(?!<` as identical to `(?<!`, or negative lookbehind. But lookahead is what we need here. Escaping the < takes care of the problem.

Continuing at The End of The Previous Match

The anchor `\G` matches at the position where the previous match ended. During the first match attempt, `\G` matches at the start of the string in the way `\A` does.

Applying `\G\w` to the string `test string` matches `t`. Applying it again matches `e`. The 3rd attempt yields `s` and the 4th attempt matches the second `t` in the string. The fifth attempt fails. During the fifth attempt, the only place in the string where `\G` matches is after the second `t`. But that position is not followed by a word character, so the match fails.

End of The Previous Match vs Start of The Match Attempt

With some regex flavors or tools, `\G` matches at the start of the match attempt, rather than at the end of the previous match result. This is the case with EditPad Pro, where `\G` matches at the position of the text cursor, rather than the end of the previous match. When a match is found, EditPad Pro will select the match, and move the text cursor to the end of the match. The result is that `\G` matches at the end of the previous match result only when you do not move the text cursor between two searches. All in all, this makes a lot of sense in the context of a text editor.

\G Magic with Perl

In Perl, the position where the last match ended is a "magical" value that is remembered separately for each string variable. The position is not associated with any regular expression. This means that you can use `\G` to make a regex continue in a subject string where another regex left off.

If a match attempt fails, the stored position for `\G` is reset to the start of the string. To avoid this, specify the continuation modifier `/c`.

All this is very useful to make several regular expressions work together. E.g. you could parse an HTML file in the following fashion:

```
while ($string =~ m/</g) {
  if ($string =~ m/\GB>/c) {
    # Bold
  } elsif ($string =~ m/\GI>/c) {
    # Italics
  } else {
    # ...etc...
  }
}
```

The regex in the while loop searches for the tag's opening bracket, and the regexes inside the loop check which tag we found. This way you can parse the tags in the file in the order they appear in the file, without having to write a single big regex that matches all tags you are interested in.

\G in Other Programming Langauges

This flexibility is not available with most other programming languages. E.g. in Java, the position for `\G` is remembered by the Matcher object. The Matcher is strictly associated with a single regular expression and a single subject string. What you can do though is to add a line of code to make the match attempt of the second Matcher start where the match of the first Matcher ended. `\G` will then match at this position.

-Then-Else Conditionals in Regular Expressions

A special construct `(?ifthen|else)` allows you to create conditional regular expressions. If the *if* part evaluates to true, then the regex engine will attempt to match the *then* part. Otherwise, the *else* part is attempted instead. The syntax consists of a pair of round brackets. The opening bracket must be followed by a question mark, immediately followed by the *if* part, immediately followed by the *then* part. This part can be followed by a vertical bar and the *else* part. You may omit the *else* part, and the vertical bar with it.

For the *if* part, you can use the lookahead and lookbehind constructs. Using positive lookahead, the syntax becomes `(?(?=regex)then|else)`. Because the lookahead has its own parentheses, the *if* and *then* parts are clearly separated.

Remember that the lookaround constructs do not consume any characters. If you use a lookahead as the *if* part, then the regex engine will attempt to match the *then* or *else part* (depending on the outcome of the lookahead) at the same position where the *if* was attempted.

For the *then* and *else*, you can use any regular expression. If you want to use alternation, you will have to group the *then* or *else* together using parentheses, like in

`(?(?=condition)(then1|then2|then3)|(else1|else2|else3))`. Otherwise, there is no need to use parentheses around the *then* and *else* parts.

Adding Comments to Regular Expressions

If you have worked through the entire tutorial, I guess you will agree that regular expressions can quickly become rather cryptic. Therefore, many modern regex flavors allow you to insert comments into regexes. The syntax is `(?#comment)` where "comment" is be whatever you want, as long as it does not contain a closing round bracket. The regex engine ignores everything after the `(?#` until the first closing round bracket.

E.g. I could clarify the regex to match a valid date by writing it as `(?#year)(19|20)\d\d[-/.](?#month)(0[1-9]|1[012])[-  /.](?#day)(0[1-9]|[12][0-9]|3[01])`. Now it is instantly obvious that this regex matches a date in yyyy-mm-dd format. Some software, such as RegexBuddy, EditPad Pro and PowerGREP can apply syntax coloring to regular expressions while you write them. That makes the comments really stand out, enabling the right comment in the right spot to make a complex regular expression much easier to understand.