**Mastering Lookahead and Lookbehind**

Lookarounds often cause confusion to the regex apprentice. I believe this confusion promptly disappears if one simple point is firmly grasped. It is that at the end of a lookahead or a lookbehind, the regex engine hasn't moved on the string. You can chain three more lookaheads after the first, and the regex engine still won't move. In fact, that's a useful technique.  
  
**A quick syntax reminder**  
This page digs deep into the details of lookahead and lookbehind and assumes you've already become familiar with the basic syntax, perhaps by reading the [lookaround section](https://www.rexegg.com/regex-disambiguation.html" \l "lookarounds) of the reference on (? … ) syntax. As a quick reminder before we dive in, here are the four lookarounds.

|  |  |  |
| --- | --- | --- |
| **Lookaround** | **Name** | **What it Does** |
| (?=foo) | Lookahead | Asserts that what immediately follows the current position in the string is *foo* |
| (?<=foo) | Lookbehind | Asserts that what immediately precedes the current position in the string is *foo* |
| (?!foo) | Negative Lookahead | Asserts that what immediately follows the current position in the string is not *foo* |
| (?<!foo) | Negative Lookbehind | Asserts that what immediately precedes the current position in the string is not *foo* |

**Jumping Points**  
For easy navigation, here are some jumping points to various sections of the page:  
  
✽ [Lookahead Example: Simple Password Validation](https://www.rexegg.com/regex-lookarounds.html#password)  
✽ [The Order of Lookaheads Doesn't Matter… Almost](https://www.rexegg.com/regex-lookarounds.html#order)  
✽ [Lookarounds Stand their Ground](https://www.rexegg.com/regex-lookarounds.html" \l "stand_their_ground)  
✽ [Various Uses for Lookarounds](https://www.rexegg.com/regex-lookarounds.html#uses)  
✽ [Zero-Width Matches](https://www.rexegg.com/regex-lookarounds.html#zero_width)  
✽ [Positioning the Lookaround Before or After the Characters to be Matched](https://www.rexegg.com/regex-lookarounds.html#position)  
✽ [Lookarounds that Look on Both Sides: Back to the Future](https://www.rexegg.com/regex-lookarounds.html" \l "back_to_the_future)  
✽ [Compound Lookahead and Compound Lookbehind](https://www.rexegg.com/regex-lookarounds.html#compound)  
✽ [The Engine Doesn't Backtrack into Lookarounds (They're Atomic)](https://www.rexegg.com/regex-lookarounds.html#atomic)  
✽ [Fixed-Width, Constrained-Width and Infinite-Width Lookbehind](https://www.rexegg.com/regex-lookarounds.html#width)  
✽ [Lookarounds (Usually) Want to be Anchored](https://www.rexegg.com/regex-lookarounds.html" \l "anchor)

**Lookahead Example: Simple Password Validation**

Let's get our feet wet right away with an expression that validates a password. The technique shown here will be useful for all kinds of other data you might want to validate (such as email addresses or phone numbers).  
Our password must meet four conditions:  
  
1. The password must have between six and ten word characters \w  
2. It must include at least one lowercase character [a-z]  
3. It must include at least **three** uppercase characters [A-Z]  
4. It must include at least one digit \d  
  
We'll assume we're working in a regex flavor where \d only matches ASCII digits 0 through 9, unlike .NET and Python where that token can match any Unicode digit.

With lookarounds, your feet stay planted on the string. You're just looking, not moving!

Our initial strategy (which we'll later tweak) will be to stand at the beginning of the string and **look ahead** four times—once for each condition. We'll look to check we have the right number of characters, then we'll look for a lowercase letter, and so on. If all the lookaheads are successful, we'll know the string is a valid password… And we'll simply gobble it all up with a plain .\*  
  
**Let's start with condition 1**  
A string that is made of six-to-ten word characters can be written like this: \A\w{6,10}\z  
The [\A anchor](https://www.rexegg.com/regex-anchors.html#A) asserts that the current position is the beginning of the string. After matching the six to ten word characters, the [\z anchor](https://www.rexegg.com/regex-anchors.html#z) asserts that the current position is the end of the string.  
  
Within a lookahead, this pattern becomes (?=\A\w{6,10}\z). This lookahead asserts: at the current position in the string, what follows is the beginning of the string, six to ten word characters, and the very end of the string.  
  
We want to make this assertion at the very beginning of the string. Therefore, to continue building our pattern, we want to anchor the lookahead with an \A. There is no need to duplicate the \A, so we can take it out of the lookahead. Our pattern becomes:  
\A(?=\w{6,10}\z)  
So far, we have an expression that validates that a string is entirely composed of six to ten word characters. Note that we haven't matched any of these characters yet: we have only looked ahead. The current position after the lookahead is still the beginning of the string. To check the other conditions, we just add lookaheads.

**Condition 2**  
For our second condition, we need to check that the password contains one lowercase letter. To find one lowercase letter, the simplest idea is to use .\*[a-z]. That works, but the dot-star first shoots down to the end of the string, so we will always need to backtrack. Just for the sport, can we think of something more efficient? You might think of making the star quantifier reluctant by adding a ?, giving us .\*?[a-z], but that too requires backtracking as a [lazy quantifier requires backtracking at each step](https://www.rexegg.com/regex-quantifiers.html#lazy_expensive).  
  
For this type of situation, I recommend you use something like [^a-z]\*[a-z] (or even better, depending on your engine, the [atomic](https://www.rexegg.com/regex-disambiguation.html#atomic) (?>[^a-z]\*)[a-z] or [possessive](https://www.rexegg.com/regex-quantifiers.html#possessive) version [^a-z]\*+[a-z]—but we'll discuss that in the [footnotes](https://www.rexegg.com/regex-lookarounds.html#atomictweak)). The negated character class [^a-z] is the *counterclass* of the lowercase letter [a-z] we are looking for: it matches one character that is not a lowercase letter, and the \* quantifier makes us match zero or more such characters. The pattern [^a-z]\*[a-z] is a good example of the principle of [contrast](https://www.rexegg.com/regex-style.html#contrast) recommended by the regex style guide.  
  
Let's use this pattern inside a lookahead: (?=[^a-z]\*[a-z])  
The lookahead asserts: at this position in the string (i.e., the beginning of the string), we can match zero or more characters that are not lowercase letters, then we can match one lowercase letter: [a-z]  
Our pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])  
At this stage, we have asserted that we are at the beginning of the string, and we have looked ahead twice. We still haven't matched any characters. Note that on a logical level it doesn't matter which condition we check first. If we swapped the order of the lookaheads, the result would be the same.  
  
We have two more conditions to satisfy: two more lookaheads.  
  
**Condition 3**  
For our third condition, we need to check that the password contains at least three uppercase letters. The logic is similar to condition 2: we look for an optional number of non-uppercase letters, then one uppercase letter… But we need to repeat that three times, for which we'll use the quantifier {3}.  
We'll use this lookahead: (?=(?:[^A-Z]\*[A-Z]){3})  
  
The lookahead asserts: at this position in the string (i.e., the beginning of the string), we can do the following three times: match zero or more characters that are not uppercase letters (the job of the negated character class [^A-Z] with the quantifier \*), then match one uppercase letter: [A-Z]  
Our pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})  
At this stage, we have asserted that we are at the beginning of the string, and we have looked ahead three times. We still haven't matched any characters.

**Condition 4**  
To check that the string contains at least one digit, we use this lookahead: (?=\D\*\d). Opposing \d to its counterclass \D makes good use of the [regex principle of contrast](https://www.rexegg.com/regex-style.html#contrast).  
  
The lookahead asserts: at this position in the string (i.e., the beginning of the string), we can match zero or more characters that are not digits (the job of the "not-a-digit" character class \D and the \* quantifier), then we can match one digit: \d  
Our pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})(?=\D\*\d)  
At this stage, we have asserted that we are at the beginning of the string, and we have looked ahead four times to check our four conditions. We still haven't matched any characters, but we have validated our string: we know that it is a valid password.  
  
If all we wanted was to validate the password, we could stop right there. But if for any reason we also need to match and return the entire string—perhaps because we ran the regex on the output of a function and the password's characters haven't yet been assigned to a variable—we can easily do so now.  
  
**Matching the Validated String**  
After checking that the string conforms to all four conditions, we are still standing at the beginning of the string. The five assertions we have made (the anchor \A and the four lookaheads) have not changed our position. At this stage, we can use a simple .\* to gobble up the string: we know that whatever characters are matched by the dot-star, the string is a valid password. The pattern becomes:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})(?=\D\*\d).\*  
[(direct link)](https://www.rexegg.com/regex-lookarounds.html#n-1conds)  
**Fine-Tuning: Removing One Condition**

For n conditions,  
use n-1 lookaheads

If you examine our lookaheads, you may notice that the pattern \w{6,10}\z inside the first one examines all the characters in the string. Therefore, we could have used this pattern to match the whole string instead of the dot-star .\*  
  
This allows us to remove one lookahead and to simplify the pattern to this:  
  
\A(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})(?=\D\*\d)\w{6,10}\z  
The pattern \w{6,10}\z now serves the double purpose of matching the whole string and of ensuring that the string is entirely composed of six to ten word characters.  
  
Generalizing this result, if you must check for n conditions, your pattern only needs to include n-1 lookaheads at the most. Often, you are even able to combine several conditions into a single lookahead.  
  
You may object that we were able to use \w{6,10}\z because it happened to match the whole string. Indeed that was the case. But we could also have converted any of the other three lookaheads to match the entire string. For instance, taking the lookahead (?=\D\*\d) which checks for the presence of one digit, we can add a simple .\*\z to get us to the end of the string.  
  
The pattern would have become:  
\A(?=\w{6,10}\z)(?=[^a-z]\*[a-z])(?=(?:[^A-Z]\*[A-Z]){3})\D\*\d.\*\z  
By the way, you may wonder why I bother using the \z after the .\*: shouldn't it get me to the end of the string? In general, not so: unless we're in [DOTALL mode](https://www.rexegg.com/regex-modifiers.html#dotall), the dot doesn't match line breaks. Therefore, the .\* only gets you to the end of the first line. After this, the string may have line breaks and many more line. A \z anchor ensures that after the .\* we have reached not only the end of the line, but also the end of the string.  
  
In this particular pattern, the first lookaround (?=\w{6,10}\z) already ensures that there cannot be any line breaks in the string, so the final \z is not strictly necessary.

**The Order of Lookaheads Doesn't Matter… Almost**

In our password validation pattern, since the three lookaheads don't change our position in the string, we can rearrange them in any order without affecting the overall logic.  
  
While the order of lookaheads doesn't matter on a logical level, keep in mind that it may matter for matching speed. If one lookahead is more likely to fail than the other two, it makes little sense to place it in third position and expend a lot of energy checking the first two conditions. Make it first, so that if we're going to fail, we fail early—an application of the [*design to fail* principle](https://www.rexegg.com/regex-style.html#design_to_fail) from the regex style guide.  
  
In fact, this is what we do by placing the anchor \A in first position. Since it is an assertion that doesn't consume characters, it too could swap positions with any of the lookaheads. We'll see why this is a bad idea, but first…  
  
In passing, consider that \A can be written with lookarounds: in [DOTALL mode](https://www.rexegg.com/regex-modifiers.html#dotall), where the dot matches any character including line breaks, the negative lookbehind (?<!.) asserts that what precedes the current position is not *any character*—therefore the position must be the beginning of the string. Without DOTALL mode, the negative lookbehind (?<![\D\d]) asserts the same, since [\D\d] matches one character that is either a digit or a non-digit—in other words, any character.  
  
Now imagine we set \A in fourth position, after the three lookaheads. The resulting match would be the same, but it could take a lot more time. For instance, suppose the third lookahead (whose job it is to assert that the string contains at least one digit) fails. After failing to find a match at the first position in the string, the engine advances to the second position and tries the lookaheads again, one after the other. Once more, the third lookahead is bound to fail to find a digit. After each failure, the engine will start a new match attempt starting at the next position in the string. Even when the two first lookaheads succeed (and they may fail, as the uppercase or lowercase letter they check for may have been the lone one in the string, and at a position already passed), the third lookahead will always fail to find a digit. Therefore the anchor \A is never even attempted: the pattern fails before the engine reaches that token.  
  
In contrast, when \A is first, it can only match at the first position in the string. The third lookahead still fails, but when the engine tries to match at further positions, the \A immediately fails, so the engine doesn't need to waste any more time with the lookaheads.

**Lookarounds Stand their Ground**

If I seem to be flogging a dead horse here, it's only because this point is the most common source of confusion with lookarounds. As the password validation example made clear, lookarounds stand their ground. They look immediately to the left or right of the engine's current position on the string—but do not alter that position.  
  
Therefore, do not expect the pattern A(?=5) to match the *A* in the string *AB25*. Many beginners assume that the lookahead says that "there is a *5* somewhere to the right", but that is not so. After the engine matches the *A*, the lookahead (?=5) asserts that at the current position in the string, what immediately follows is a *5*. If you want to check if there is a *5* somewhere (anywhere) to the right, you can use (?=[^5]\*5).  
  
Moreover, don't expect the pattern A(?=5)(?=[A-Z]) to match the *A* in the string *A5B*. Many beginners assume that the second lookahead looks to the right of the first lookahead. It is not so. At the end of the first lookahead, the engine is still planted at the very same spot in the string, after the *A*. When the lookahead (?=[A-Z]) tries to assert that what immediately follows the current position is an uppercase letter, it fails because the next character is still the *5*. If you want to check that the 5 is followed by an uppercase letter, just state it in the first lookahead: (?=5[A-Z])  
  
So *lookahead* and *lookbehind* don't mean "look way ahead into the distance". They mean "look at the text immediately to the left or to the right". If you want to inspect a piece of string further down, you will need to insert "binoculars" inside the lookahead to get you to the part of the string you want to inspect—for instance a .\*, or, ideally, more specific tokens.

**Various Uses for Lookarounds**

Before we dive into interesting but sometimes terse details, let's get excited about lookarounds by surveying some of their terrific uses.

**Validation**  
The [password validation](https://www.rexegg.com/regex-lookarounds.html#password) section showed how the combination of several lookaheads can impose a number of conditions on the string to be matched, allowing us to validate it with a single pattern.  
  
**Restricting a Character Range (Subtraction, Intersection)**  
Suppose you want to match one word character \w as long as it is not the letter *Q*. There are several ways to do it without lookarounds:  
✽ In engines that support [character class subtraction](https://www.rexegg.com/regex-class-operations.html#subtraction), you can use [\w-[Q]] (.NET), [\w&&[^Q]] (Java and Ruby 1.9+) or [\w--Q] (Python with the alternate *regex* module)  
✽ You can build a character class such as [\_0-9a-zA-PR-Z]  
✽ You can use [^\WQ]—an example of an [obnoxious double-negative character range](https://www.rexegg.com/regex-interesting-character-classes.html#obnoxious).  
  
If your engine doesn't support character class subtraction, the simplest may be to use the [workaround](https://www.rexegg.com/regex-class-operations.html#subtraction_workaround) shown on the page about class operations. This uses a lookahead to restrict the character class \w:  
(?!Q)\wAfter the negative lookahead asserts that what follows the current position is not a *Q*, the \w matches a word character.  
  
Not only is this solution easy to read, it is also easy to maintain if we ever decide to exclude the letter *K* instead of *Q*, or to exclude both: (?![QK])\w  
  
Note that we can also perform the same exclusion task with a negative lookbehind:  
\w(?<!Q)After the \w matches a word character, the negative lookbehind asserts that what precedes the current position is not a *Q*.  
  
Using the same idea, if we wanted to match one character in the Arabic script as long as it is not a number, we could use this pattern:  
(?!\p{N})\p{Arabic}This would work in Perl, PCRE (C, PHP, R…) and Ruby 2+. In .NET and Java, you would use (?!\p{N})\p{IsArabic}  
  
Likewise, we can use this technique to perform a DIY [character class intersection](https://www.rexegg.com/regex-class-operations.html#intersection). For instance, to match one character in the Arabic script as long as it is a number, we transform the negative lookahead above to a positive lookahead. In the Perl / PCRE / Ruby version, this gives us:  
(?=\p{N})\p{Arabic}  
This is basically the [password validation technique](https://www.rexegg.com/regex-lookarounds.html#password) with two conditions applied to a single character.  
  
Needless to say, you can interchange the content of the lookahead with the token to be matched:(?=\p{Arabic})\p{N}  
**Tempering the scope of a token**  
This use is similar to the last. Instead of removing characters from a class, it restricts the scope within which a token is allowed to match.  
  
For instance, suppose we want to match any character as long as it is not followed by *{END}*. Using a negative lookahead, we can use:  
(?:(?!{END}).)\*Each . token is tempered by (?!{END}), which specifies that the dot cannot be the beginning of *{END}*. This technique is called [*tempered greedy token*](https://www.rexegg.com/regex-quantifiers.html#tempered_greed) on the *Quantifiers page*.  
  
Another technique is:  
(?:[^{]++|{(?!END}))\*+On the left side of the alternation, [^{]++ matches characters that are not an opening brace. On the right side, {(?!END}) matches an opening brace that is not followed by *END}*. This technique appears in the [*Explicit Greedy Alternation*](https://www.rexegg.com/regex-quantifiers.html#explicit_greed) section of the *Quantifiers page*.  
  
**Delimiter**  
Do you have a string where you want to start matching all characters once the first instance of *#START#* is passed? No problem, just use a lookbehind to make a delimiter:  
(?<=#START#).\*After the lookbehind asserts that what immediately precedes the current position is *#START#*, the dot-star .\* matches all the characters to the right.  
  
Or would you like to match all characters in a string up to, but not including the characters *#END#*? Make a delimiter using a lookahead:  
.\*?(?=#END#)  
You can, of course, combine the two:  
(?<=#START#).\*?(?=#END#)  
See the page on boundaries for advice on building fancy [DIY delimiters](https://www.rexegg.com/regex-boundaries.html#diy).  
  
[(direct link)](https://www.rexegg.com/regex-lookarounds.html#camelinsert)  
**Inserting Text at a Position**  
Someone gave you a file full of film titles in CamelCase, such as *HaroldAndKumarGoToWhiteCastle*. To make it easier to read, you want to insert a space at each position between a lowercase letter and an uppercase letter. This regex matches these exact positions:  
(?<=[a-z])(?=[A-Z])  
In your text editor's regex replacement function, all you have to do is replace the matches space characters, and spaces be inserted in the right spot.  
  
This regex is what's known as a "zero-width match" because it matches a position without matching any actual characters. How does it work? The lookbehind asserts that what immediately precedes the current position is a lowercase letter. And the lookahead asserts that what immediately follows the current position is an uppercase letter.  
  
**Splitting a String at a Position**  
We can use the exact same regex from the previous example to split the string *AppleOrangeBananaStrawberryPeach* into a list of fruits. Once again, the regex  
(?<=[a-z])(?=[A-Z])matches the positions between a lowercase letter and an uppercase letter.  
  
In most languages, when you feed this regex to the function that uses a regex pattern to split strings, it returns an array of words.  
  
Note that Python's *re* module does not split on zero-width matches—but the far superior *regex* module does.  
  
  
**Finding Overlapping Matches**  
Sometimes, you need several matches within the same word. For instance, suppose that from a string such as *ABCD* you want to extract *ABCD*, *BCD*, *CD* and *D*. You can do it with this single regex:  
(?=(\w+))When you allow the engine to find all matches, all the substrings will be captured to Group 1  
  
How does this work?  
  
At the first position in the string (before the *A*), the engine starts the first match attempt. The lookahead asserts that what immediately follows the current position is one or more word characters, and captures these characters to Group 1. The lookahead succeeds, and so does the match attempt. Since the pattern didn't match any actual characters (the lookahead only looks), the engine returns a zero-width match (the empty string). It also returns what was captured by Group 1: *ABCD*  
  
The engine then moves to the next position in the string and starts the next match attempt. Again, the lookahead asserts that what immediately follows that position is word characters, and captures these characters to Group 1. The match succeeds, and Group 1 contains *BCD*.  
  
The engine moves to the next position in the string, and the process repeats itself for *CD* then *D*.  
  
In .NET, which has infinite lookbehind, you can find overlapping matches from the other side of the string. For instance, on the same string *ABCD*, consider this pattern:  
(?<=(\w+))  
It will capture *A*, *AB*, *ABC* and *ABCD*. To achieve the same in an engine that doesn't support infinite lookbehind, you would have to reverse the string, use the lookahead version (?=(\w+)) then reverse the captures.

**Zero-Width Matches**

As we've seen, a lookaround looks left or right but it doesn't add any characters to the match to be returned by the regex engine. Likewise, an anchor such as ^ and a boundary such as \b can match at a given position in the string, but they do not add any characters to the match.  
  
Usually, lookaheads, lookbehinds, anchors and boundaries appear in patterns that contain tokens that do match characters, allowing the engine to return a matched string. For instance, in (?<=start\_)\d+, the engine matches and returns some digits, but not the prefix *start\_*  
  
However, if a pattern only contains lookarounds, anchors and boundaries, the engine may be able to match the pattern without matching any characters. The resulting match is called a *zero-width match* because it contains no characters.  
  
This can be a useful technique, and we have already seen some applications of zero-width matches in the section on [uses for lookarounds](https://www.rexegg.com/regex-lookarounds.html#uses). To bring them together under one heading, here are some of their main uses.  
  
**Validation**  
If you string several lookarounds in a row, you can validate that a string conforms to a set of rules, as in the [password validation technique](https://www.rexegg.com/regex-lookarounds.html#password).  
  
We saw that when you have n conditions, if you also want to match the string, you usually need n-1 lookarounds at the most as [one condition can be removed](https://www.rexegg.com/regex-lookarounds.html#n-1conds) and used in the matching section of the pattern. But if all you want to do is validate, all the conditions can stay inside lookarounds, giving you a zero-width match.  
  
**Inserting**  
You can use a zero-width match regex to match a position in a string and insert text at that position. For instance, by matching (?m)^ (the beginning of a line in multiline mode) and replacing the match with // , you can add a prefix to every line of a file.  
  
Likewise, we saw how the zero-width pattern (?<=[a-z])(?=[A-Z]) allows you to [insert characters in a CamelCase word](https://www.rexegg.com/regex-lookarounds.html#camelinsert).  
  
**Splitting**  
We saw how the same zero-width pattern (?<=[a-z])(?=[A-Z]) allows you to [split a CamelCase word](https://www.rexegg.com/regex-lookarounds.html#camelsplit) into its components.  
  
**Overlapping Matches**  
We saw how an unanchored lookaround that contains capture groups—such as (?=(\w+))—allows you to [match overlapping string segments](https://www.rexegg.com/regex-lookarounds.html#overlapping).

**Positioning the Lookaround**

Often, you have two options for positioning a lookaround: before the text to be matched, or after. Usually, one of the options is more efficient because it requires less work of the engine.  
  
To illustrate this, here are examples for each kind of lookaround. I borrowed them from the [lookarounds section](https://www.rexegg.com/regex-disambiguation.html" \l "lookarounds) of the main syntax page, where they are discussed in greater detail.  
  
**Lookahead**  
\d+(?= dollars) and (?=\d+ dollars)\d+ both match *100* in 100 dollars, but the first is more efficient because the engine needs to match \d+ only once.  
  
**Negative Lookahead**  
\d+(?! dollars) and (?!\d+ dollars)\d+ both match *100* in 100 pesos, but the first is more efficient because the engine needs to match \d+ only once.  
  
**Lookbehind**  
(?<=USD)\d{3} and \d{3}(?<=USD\d{3}) both match *100* in USD100, but the first is more efficient because the engine needs to match \d{3} only once.  
  
**Negative Lookbehind**  
(?<!USD)\d{3} and \d{3}(?<!USD\d{3}) both match *100* in JPY100, but the first is more efficient because the engine needs to match \d{3} only once.  
  
What may not be so clear is that each of these lookarounds can be used in two main ways: before the expression to be matched, or after it. These two ways have a slightly different feel. Please don't obsess over the differences; rather, just cruise through these simple examples to become familiar with the types of effects you can achieve.  
  
When you compare each pair, the two methods have a different *feel*. The point of the examples is not to make you memorize "the right position", but to expose you to those two basic feels. Once you're familiar with them, you will naturally think of rewriting a lookaround that feels too heavy. With a bit of practice, the efficient way of positioning your lookarounds will probably come to you naturally.

**Lookarounds that Look on Both Sides: Back to the Future**

Suppose you want to match a two-digit number surrounded by underscores as in \_12\_ but not the underscores.  
  
We have already seen three ways to do this:  
✽ You can match everything and capture the digits to Group 1: \_(\d{2})\_  
✽ You can use a lookbehind and a lookahead: (?<=\_)\d{2}(?=\_)  
✽ You can use \K to drop the first underscore from the match: \_\K\d{2}(?=\_)  
  
There is a fourth technique I'd like to introduce you to. I call it the "back to the future lookbehind." There shouldn't be any reason to use it on its own, but sometimes within an intricate pattern it may just what you need, so it's nice to be familiar with it and add it to your repertoire.  
  
We can position our back-to-the-future lookbehind before or after the digits. Let's start with the *before* version:  
(?<=\_(?=\d{2}\_))\d+  
Wowzy, what does this do? The lookbehind asserts that what immediately precedes the current position in the string is an underscore, then a position where the lookahead (?=\d{2}\_) can assert that what immediately follows is two digits and an underscore.  
  
This is interesting for several reasons. First, we have a lookahead within a lookbehind, and even though we were supposed to look backwards, this lookahead jumps over the current position by matching the two digits and the trailing underscore. That's acrobatic.  
  
Second, note that even though it looks complex, this is a fixed-width lookbehind (the width is one character, the underscore), so it should work in all flavors of lookbehind. (However, it does not work in Ruby as Ruby does not allow lookaheads and negative lookbehinds inside lookbehind.)  
  
Another interesting feature is how the notion of "current position in the string" is not the same for the lookbehind and for the lookahead. You'll remember that [lookarounds stand their ground](https://www.rexegg.com/regex-lookarounds.html" \l "stand_their_ground), so that after checking the assertion made by a lookaround, the engine hasn't moved in the string. Are we breaking that rule?  
  
We're not. In the string *10 \_16\_ 20*, let's say the engine has reached the position between the underscore and the *1* in *16*. The lookbehind makes an assertion about what can be matched at that position. When the engine exits the lookbehind, it is still standing in that same spot, and the token \d{2} can proceed to match the characters *16*.  
  
But within the lookbehind itself, we enter a different little world. You can imagine that outside that world the engine is red, and inside the little world of the lookbehind, there is another little engine which is yellow. That yellow engine keeps track of its own position in the string. In most engines (.NET proceeds differently), the yellow engine is initially dropped at a position in the string that is found by taking the red engine's position and subtracting the width of the lookbehind, which is 1. The yellow engine therefore starts its work before the leading underscore. Within the lookbehind's little world, after matching the underscore token, the yellow engine's position in the string is between the underscore and the *1*. It is that position that the lookahead refers to when it asserts that at the current position in the string (according to the little world of the lookbehind and its yellow engine), what immediately follows is two digits and an underscore.  
  
**After the digits**  
Here is a second version where the "back-to-the-future lookbehind" comes after the digits:  
\d+(?<=\_\d{2}(?=\_))  
The lookbehind states: what immediately precedes this position in the string is an underscore and two digits, then a position where the lookahead (?=\_) can assert that what immediately follows the current position in the string (according to the yellow engine and the lookbehind's little world) is an underscore.  
  
This too is a fixed-width lookbehind (the width is three character, i.e. the leading underscore and the two digits), so it should work in all flavors of lookbehind except Ruby.  
  
  
**Compound Lookahead and Compound Lookbehind**

The [back-to-the-future lookbehind](https://www.rexegg.com/regex-lookarounds.html#back_to_the_future) introduced us to what I call *compound lookarounds*, i.e., lookarounds that contain other lookarounds. You could also call them *nested lookarounds*, but for me the idea of compounding captures something more about the feel of working with these constructs.  
  
Let's look at some examples.  
  
**Token followed by one character, but not more**  
How can you match a number that is followed by one underscore, but not more?  
  
You can use this:  
\d+(?=\_(?!\_))The lookahead asserts: what follows the current position in the string is one underscore, then a position where the negative lookahead (?!\_) can assert that what follows is not an underscore. A less elegant variation would be \d+(?=(?!\_\_)\_)  
  
**Token preceded by one character, but not more**  
How can you match a number that is preceded by one underscore, but not more?  
  
You can use this:  
(?<=(?<!\_)\_)\d+The lookbehind asserts: what precedes the current position in the string is a position where the negative lookbehind (?<!\_) can assert that what immediately precedes is not an underscore, then an underscore. A variation would be (?<=\_(?<!\_\_))\d+  
  
**Multiple Compounding**  
Needless to say, it won't be long until you find occasions to add levels of compounding beyond the two we've just seen. But that quickly becomes obnoxious, and it becomes simpler to rearrange the regex. For instance, building on the previous pattern,  
(?<=(?<!(?<!X)\_)\_)\d+matches a number that is precede by an underscore that is not preceded by an underscore unless that underscore is preceded by an *X*.  
  
In .NET, PCRE, Java and Ruby, this could be simplified to (?<=(?<!\_)\_|X\_\_)\d+  
In Perl and Python, you could use (?:(?<=(?<!\_)\_)|(?<=X\_\_))\d+

**The Engine Doesn't Backtrack into Lookarounds…  
  
…because they're atomic**

Here's a fun regex task. You have a string like this:  
\_rabbit \_dog \_mouse DIC:cat:dog:mouse  
The *DIC* section at the end contains a list of allowed animals. Our job is to match all the *\_tokens* named after an allowed animal. Therefore, we expect to match *\_dog* and *\_mouse*. A lookaround helps us do this:  
  
\_(\w+)\b(?=.\*:\1\b)  
After matching the underscore, we capture a word to Group 1. Then the lookahead (?=.\*:\1\b) asserts what follows the current position in the string is zero or more characters, then a colon, then the word captured to Group 1. As hoped, this matches both *\_dog* and *\_mouse*.  
  
Now suppose we try a "reversed" approach:  
  
\_(?=.\*:(\w+)\b)\1\b  
This only matches *\_mouse*. Why?  
  
First let's try to understand what this regex hopes to accomplish. It may not be that obvious, but it illustrates an important feature of lookarounds.  
  
After the engine matches the underscore, the lookahead (?=.\*:(\w+)\b) asserts that what follows the current position in the string is any number of characters, then a colon, then a word (captured to Group 1). After passing that assertion, the back-reference \1 matches what was captured into Group 1.  
  
Let's see how this works out. Remember that our string is  
\_rabbit \_dog \_mouse DIC:cat:dog:mouse  
After the underscore that precedes *rabbit*, we expect the lookahead to fail because there is no *rabbit* in the *DIC* section—and it does. The next time we match an underscore is before *dog*. At that stage, inside the lookahead (?=.\*:(\w+)\b), the dot-star shoots down to the end of the string, then backtracks just far enough to allow the colon to match, after which the word *mouse* is matched and captured to Group 1. The lookahead succeeds. The next token \1 tries to match *mouse*, but the next character in the string is the *d* from *dog*, so the token fails. At this stage, having learned everything about backtracking, we might assume that the regex engine allows the dot-star to backtrack even more inside the lookahead, up to the previous colon, which would then allow (\w+) to match and capture *mouse*. Then the back-reference \1 would match *mouse*, and the engine would return a successful match.  
  
However, it does not work that way. Once the regex engine has left a lookaround, it will not backtrack into it if something fails somewhere down the pattern. On a logical level, that is because the official point of a lookaround is to return one of two values: true or false. Once a lookahead evaluates to true at a given position in the string, it is always true. From the engine's standpoint, there is nothing to backtrack. What would be the point—since the only other available value is false, and that would fail the pattern?  
  
The fact that the engine will not backtrack into a lookaround means that it is an [atomic block](https://www.rexegg.com/regex-disambiguation.html#atomic). This property of lookarounds will rarely matter, but if someday, in the middle of building an intricate pattern, a lookahead refuses to cooperate… This may be the reason.

**Fixed-Width, Constrained-Width and Infinite-Width Lookbehind**

In strings such as *123456\_ORANGE abc12\_APPLE*, suppose you are interested in matching uppercase words, provided they are preceded by a prefix composed of digits and an underscore character. Therefore, in this string, you want to match *ORANGE* but not *APPLE*.  
  
It's worth remembering that in most regex flavors (.NET is one of the few exceptions), the following pattern is invalid:  
  
(?<=\b\d+\_)[A-Z]+  
That is because the width of the text matched by the token \d+ can be anything. Most engines require the width of the subexpression within a lookbehind to be known in advance, as in (?<=\d{3})  
  
Some engines allow the width of the subexpression within a lookbehind to take various pre-determined values found on the various sides of an alternation, as in (?<=0|128|\d{6}). Yet others allow the width to vary within a pre-determined range, as in (?<=d{2,6})  
  
For details of what kinds of widths various engines allow in a lookbehind, see the [Lookbehind: Fixed-Width / Constrained Width / Infinite Width](https://www.rexegg.com/regex-disambiguation.html#lookbehind_width) section of the main syntax page. To honor the winners, I'll just repeat here that the only two programming-language flavors that support infinite-width lookbehind are .NET (C#, VB.NET, …) and Matthew Barnett's [*regex* module](https://pypi.python.org/pypi/regex) for Python. I've also implemented an [infinite lookbehind demo for PCRE](https://www.rexegg.com/pcre-callouts.html#infinite_lb).  
  
**Capture Group Inside Variable Lookbehind: Difference between Java and .NET**  
Both Java and .NET allow this pattern:  
(?<=(\d{1,5}))Z  
.NET allows it because it supports infinite-width lookbehind. Java allows it because it supports lookbehind whose width falls within a defined range. However, they operate differently. As a result, against the string ***123Z***, this pattern will return different Group 1 captures in the two engines.  
  
✽ Java captures *3* to Group 1. The engine sees that the width of the string to be matched inside the lookbehind must fall between one and five characters. Java tries all the possible fixed-width patterns in the range, from the shortest to the longest, until one succeeds. The shortest possible fixed-width pattern is (?<=(\d{1})). The engine temporarily skips back one character in the string, tries to match \d{1} and succeeds. The lookaround succeeds, and Group 1 contains *3*.  
  
✽ .NET captures *123* to Group 1. The .NET engine has a far more efficient way of processing variable-width lookbehinds. Instead of trying multiple fixed-width patterns starting at points further and further back in the string, .NET reverses the string as well as the pattern inside the lookbehind, then attempts to match that single pattern on the reversed string. Therefore, in *123Z*, to try the lookbehind at the point before *Z*, it reverses the portion of string to be tested from *123* to *321*. Likewise, the lookbehind (?<=(\d{1,5})) is flipped into the lookahead (?=(\d{1,5})). \d{1,5} matches *321*. Reversing that string, Group 1 contains *123*. To only capture *3* as in Java, you would have to make the quantifier lazy: (?<=(\d{1,5}?))Z  
  
✽ Like .NET, the *regex* alternate regular expressions module for Python captures *123* to Group 1.  
  
  
**Workarounds**  
There are two main workarounds to the lack of support for variable-width (or infinite-width) lookbehind:  
  
✽ Capture groups.  
Instead of (?<=\b\d+\_)[A-Z]+ , you can use \b\d+\_([A-Z]+), which matches the digits and underscore you don't want to see, then matches and captures to Group 1 the uppercase text you want to inspect. This will work in all major regex flavors.  
  
✽ The \K "keep out" verb, which is available in Perl, PCRE (C, PHP, R…), Ruby 2+ and Python\'s alternate [*regex* engine](https://pypi.python.org/pypi/regex).  
\K tells the engine to drop whatever it has matched so far from the match to be returned. Instead of (?<=\b\d+\_)[A-Z]+, you can therefore use \b\d+\_\K[A-Z]+  
  
Compared with lookbehinds, both the \K and capture group workarounds have limitations:  
  
✽ When you look for multiple matches in a string, at the starting position of each match attempt, a lookbehind can inspect the characters behind the current position in the string. Therefore, against ***123***, the pattern (?<=\d)\d (match a digit preceded by a digit) will match both *2* and *3*. In contrast, *\d\K\d* can only match *2*, as the starting position after the first match is immediately before the *3*, and there are not enough digits left for a second match. Likewise, \d(\d) can only capture *2*.  
  
✽ With lookbehinds, you can impose multiple conditions (similar to our [password validation technique](https://www.rexegg.com/regex-lookarounds.html#password)) by using multiple lookbehinds. For instance, to match a digit that is preceded by a lower-case Greek letter, you can use (?<=\p{Ll})(?<=\p{Greek})\d. The first lookbehind (?<=\p{Ll}) ensures that the character immediately to the left is a lower-case letter, and the second lookbehind (?<=\p{Greek}) ensures that the character immediately to the left belongs to the Greek script. With the workarounds, you could use \p{Greek}\K\d to match a digit preceded by a character in the Greek script (or \p{Greek}(\d) to capture it), but you cannot impose a second condition. To get over this limitation, you could capture the Greek character and use a second regex to check that it is a lower-case letter.  
  
  
[(direct link)](https://www.rexegg.com/regex-lookarounds.html#anchor)

**Lookarounds (Usually) Want to be Anchored**

Let's imagine we want to match a string consisting of one word, provided it contains at least one digit. This pattern offers a reasonable solution—one of several:  
\A(?=\D\*\d)\w+\z  
The [\A anchor](https://www.rexegg.com/regex-anchors.html#A) asserts that the current position is the beginning of the string. The lookahead (?=\D\*\d) asserts that at the current position (which is still the beginning of the string), we can match zero or more non-digits, then one digit. Next, \w+ matches our word. Finally, the [\z anchor](https://www.rexegg.com/regex-anchors.html#z) asserts that the current position is the end of the string.  
  
Now consider what happens when we forget the anchor \A and use (?=\D\*\d)\w+\z. To make our oversight seem less severe, let's assume we know that our string always contains an uninterrupted string of word characters. This guarantees that if we find a match, it will have to be the right one—at the beginning of the string, as we wanted. So what's the problem?  
  
Suppose we use our regex on a string composed of one hundred characters *V*. Since the string doesn't contain a single digit, you and I can immediately see that the regex must fail. Let's see how fast the engine comes to the same conclusion.  
  
As always, the engine begins by trying to match the pattern at the first position in the string. Starting with the first token (?=\D\*\d), it tries to assert that at the current position, i.e. the beginning of the string, it can match zero or more non-digits, then one digit. Within the subexpression, the \D\* matches all the *V* characters. The engine then tries to match a digit, but since we have reached the end of the string, that fails.  
  
If we're using a smart engine such as PCRE, at this stage the engine fails the lookaround for this first match attempt. That's because before starting the match attempt, the engine has studied the pattern and noticed that the \D and \d tokens are mutually exclusive, and it has turned the \* quantifier into a possessive quantifier \*+, a process known to PCRE as *auto-possessification* (see [footnote](https://www.rexegg.com/regex-lookarounds.html#atomic)).  
  
A less clever engine will backtrack, giving up all the \D characters it has matched one by one, each time attempting to match a \d after giving up a \D. Eventually, the engine runs out of characters to backtrack, and the lookahead fails.  
  
Once the engine understands that the lookahead must fail (whether it comes to this conclusion cleverly or clumsily), it gives up on the entire first match attempt. Next, as always in such cases, the engine moves to the next position in the string (past the first *V*) and starts a new match attempt. Again, the \D\* eats up all the *V* characters—although this time, there are only 99 of them. Again, the lookahead fails, either fast if the engine is smart, or, more likely, after backtracking all the way back to the starting position.  
  
After failing a second time, the engine moves past the second *V*, starts a new match attempt, and fails… And so on, all the way to the end of the string.  
  
Because the pattern is not anchored at the beginning of the string, at each match attempt, the engine checks whether the lookahead matches at the current position. In doing so, in the best case, it matches 100 *V* characters, then 99 on the second attempt, and so on—so it needs about 5000 steps before it can see that the pattern will never match. In the more usual case, the engine needs to backtrack and try the \d at each position, adding two steps at each *V position*. Altogether, it needs about 15,000 steps before it can see that the pattern will never match.  
  
In contrast, with the original anchored pattern \A(?=\D\*\d)\w+\z, after the engine fails the first match attempt, each of the following match attempts at further positions in the string fail instantly, because the \A fails before the engine gets to the lookahead. In the best case, the engine takes about 200 steps to fail (100 steps to match all the *V* characters, then one step at each of the further match attempts.) In the more usual case, the engine takes about 400 steps to fail (300 steps on the first match attempt, then one step at each of the further match attempts.)  
  
Needless to say, the ratio of (15,000 / 400) steps is the kind of performance hit we try to avoid in computing. This makes a solid case for helping the engine along by minimizing the number of times lookaheads must be attempted, either by using [anchors](https://www.rexegg.com/regex-anchors.html) such as ^ and \A, or by matching literal characters immediately before the lookahead.  
  
**One Exception: Overlapping Matches**  
There are times when we do want the engine to attempt the lookahead at every single position in the string. Usually, the purpose of such a maneuver is to match a number of overlapping substrings. For instance, against the string word, if the regex (?=(\w+)) is allowed to match repeatedly, it will match four times, and each match will capture a different string to Group 1: *word*, *ord*, *rd*, then *d*. The section on [overlapping matches](https://www.rexegg.com/regex-lookarounds.html#overlapping) explains how this works.

**Footnotes**

**Atomic tweak**  
The [atomic](https://www.rexegg.com/regex-disambiguation.html#atomic) variation (?>[^a-z]\*)[a-z] or [possessive](https://www.rexegg.com/regex-quantifiers.html#possessive) version [^a-z]\*+[a-z] are tweaks that ensure that if the engine fails to find the lowercase letter, it won't "stupidly" backtrack, giving up the non-lowercase letters one by one to see if a lowercase letter might fit at that stage.  
  
Note that before they start matching, some engines notice the mutually exclusive character of [a-z] and its counterclass and automatically make the \* quantifier possessive for you. This optimization is what PCRE calls *auto-possessification*. It allows you to turn it off with the [Special Start-of-Pattern Modifier](https://www.rexegg.com/regex-modifiers.html#pcre) (\*NO\_AUTO\_POSSESS)—but why would you ever want to?