Regular Expressions

Notes by Jan Marek

Regular expressions - Notes by Jan Marek

Table Of Contents

Regular expressions

- Working with text is what coders do. Sure, it's carefully structured text that fits whatever syntax your preferred programming language uses, but it's still lots and lots of text. And then there's the command line: a whole world of power, all controlled through pure text.
- Regular expressions – known as regexes and pronounced "rej exiz" – are a specialized syntax that let you search and replace text using match criteria. When you were using * and ? on the terminal to match filenames, you can think of them as very simple regexes: * means "match anything" and ? means "match any one thing."
- Every development environment worth its salt has support for regexes baked right in. Whether you use WebStorm or Visual Studio, Xcode or Atom, they all support regexes. This isn't a coincidence: they are a tool that can turn hours, days, or even weeks of work into seconds of computer time, and they are a highly prized asset in any developer's toolbox.
- There are some downsides to regex. First, they are designed to solve the problem of parsing text that follows a specific pattern. If you're trying to parse something extremely complicated – raw HTML being the usual offender –then regexes are the wrong solution. Second, even moderately difficult regexes can start to look like line noise rather than something meaningful, so you really need to learn them in a graded way to avoid being overwhelmed.

Character classes

- Let’s start with the most fundamental type of regular expression: the character class. This lets you specify a group of letters that should be matched, either by specifically listing each of them or by using a character range.
- Let’s start with something simple: enter the test string “the cat sat on the mat” into regex101.com. We can use regular expressions to match “cat”, “sat”, and “mat” in a single expression, all be using character classes. Enter this into the regular expression field:
    - [csm]at
- That’s our first regex, and it means “a ‘c’, an ’s’, or an ‘m’, followed by ‘at’.” That will match cat, sat, and mat, but not bat, fat, hat, and so on. Regexes are designed to stop processing text as soon as a match is found – called “eager” – so that they avoid wasting CPU time, so on Regex101 you’ll see only the word “cat” highlighted.
- We want to enable “global” mode, which forces the regex engine to find all matches, so please type the letter “g” into the box to the right of the regex entry – that’s the list of regex modifiers. You should now see “cat”, “sat”, and “mat” highlighted.
- You’ve seen how [cm]at matches “cat” and “mat”, but in place of [cm] you can write [a- z] to match any lowercase letter. So, [a-z]at will match everything that [csmfh]at did, but also rat, eat, oat, pat, and so on – the regex engine automatically converts [a-z] into the full lowercase alphabet. Of course, it will also match “aat” which isn’t a real word.
- Regardless of which approach you use, the regex will look for matches anywhere inside your text: it doesn’t matter whether the word is “eat”, “heat”, “heater”, or even “amphitheaters” – as long as there’s a lowercase letter followed by “at” somewhere in there, the regex will match those three letters. In doing so, it will blithely ignore the word it’s attached to: if your text string really is “amphitheaters” then only the “eat” part will be matched.
- Regex ranges don’t have to be the full alphabet if you prefer. You can use [a-t] to exclude the letters U to Z.

Case sensitivity

- Regexes are case-sensitive by default, which means “Cat” and “Mat” won’t be matched. Ignoring case insensitivity depends on how you’re using regular expressions:
    - • In Regex101 you need to add the letter “i” to the regex modifiers, where we have “g” right now.
    - • If you’re using a text editor, disable case-sensitive search.
    - • If you’re using PHP or JavaScript, regexes are written as /[cm]at/, and you need to add “i” after the final slash, like this: /[cm]at/i.
    - • If you’re using Java, you need to call Pattern.compile() with the flag Pattern.CASE_INSENSITIVE.
    - • If you’re using C#, you need to create your Regex object with the flag RegexOptions.IgnoreCase.
    - • If you’re using Objective-C, you need to create your NSRegularExpression object with the flag NSRegularExpressionCaseInsensitive.
    - • If you’re using Swift, you need to create your NSRegularExpression object with the flag .CaseInsensitive.
- Alternatively, you can just add more things to your regex. We used [a-z]at to match cat, hat, mat, and so on, but we can make it also match Cat, Hat, and Mat like this: [A-Za-z]at. That now reads “any uppercase or lowercase letter followed by ‘at’”, which is exactly what we want. You can mix uppercase and lowercase letters without ranges if you want, for example, [Cc]at will match only “Cat” and “cat”. You can also mix ranges with specific characters, for example [Ca-z] would match “cat”, “sat”, “mat”, and also “Cat” – but not “Hat”, or “Mat”, because only an uppercase C is allowed.
- As well as the uppercase and lowercase ranges, you can also specify digit ranges with character classes. This is most commonly [0-9] to allow any number, or [A-Za-z0-9] to allow any alphanumerical letter, but you might also use [A-Fa-f0-9] to match hexadecimal numbers, for example.
- Using what you know, let’s examine variables in programming languages. These usually have a specific format no matter what language you use: you can use any combination of letters, numbers, or underscores, but the first letter must be a letter or underscore. This requires using two character classes back to back, for example this regex will match variables like “aa”, “a9”, and “_i”:
    - [A-Za-z_][A-Za-z0-9_]
- if your character class starts with a caret – that’s the ^ symbol – it inverts the set, i.e. it means the opposite.
    - [^a-z]at
    - The regex [^a-z]at means “anything that isn’t a lowercase letter, followed by ‘at’.” This means “fat”, “cat”, “sat”, and “mat” won’t match, but “Pat” will.

Quantification

- We created the regex [A-Za-z_][A-Za-z0-9_] to match variable names, which means the test string must start with a letter of any case or an underscore, followed by a letter of any case or a number or an underscore. That works great if you name all your variables i, aa, or similar, but let's face it: you probably don't.
- Instead, we name variables like dataStore and members, which means we want to be able to match strings of any length. To make this work, I need to introduce a new regex concept called quantification: the ability to say how many times something ought to appear.
- In this case, we're going to use the asterisk quantifier, *. This means "zero or more matches", and allows us to say that variable names must start with an uppercase or lowercase letter, then zero or more uppercase letters, lowercase letters, numbers, or an underscore. Quantifiers always appear after the thing they are modifying, so the regex is this:
    - [A-Za-z_][A-Za-z0-9_]*
    - That's it: just by adding a * to the end we now match the convention used to name variables in most programming languages.
- As well as *, there are two other similar quantifiers: + and ?. If you use + it means "one or more", which is ever so slightly different from the "zero or more" of *. And if you use ? it means "zero or one."
- These quantifiers are really fundamental in regexes, so I want to make sure you really understand the difference between them. So, consider these three regexes:
    - 1. [A-Za-z_][A-Za-z0-9_]*
    - 2. [A-Za-z_][A-Za-z0-9_]+
    - 3. [A-Za-z_][A-Za-z0-9_]?
- I want you to look at each of those three regexes, then consider this: when given the test string "myVariable" what will each of those three match? What about when given the test string "i"?
    - The first regex [A-Za-z_][A-Za-z0-9_]* means "any uppercase letter, lowercase letter, or underscore, followed by zero or more uppercase letters, lowercase letters, digits, or underscores." So, when given "myVariable" it should match the entire test string: "myVariable". When given "i" it should also match "i".
    - The second regex [A-Za-z_][A-Za-z0-9_]+ means "any uppercase letter, lowercase letter, or underscore, followed by one or more uppercase letters, lowercase letters, digits, or underscores." This is subtly different from using *: "myVariable" will still be matched, but "i" will not because it doesn't contain one or more of the second character class.
    - The third regex [A-Za-z_][A-Za-z0-9_]? means "any uppercase letter, lowercase letter, or underscore, followed by zero or one uppercase letters, lowercase letters, digits, or underscores." When given the test string "myVariable" this will only match the first two letters, "my", and when given "i" it will match "i".

- Quantifiers aren’t just restricted to character classes. For example, if you wanted to match the word “color” in both US English (“color”) and International English (“colour”), you could use the regex colou?r. That is, “match the exact string ‘colo’, the match zero or one ‘u’s, then an ‘r’.”

Matching specific quantities

- You’ve seen how * means “match zero or more” and + means “match one or more”, but you can also be more specific: “I want to match exactly three characters.” This is done using braces, { and }.
  - For example [a-z]{3} means “match exactly three lowercase letters.”
- Consider a phone number formatted like this: 111-1111. We want to match only that format, and not “11-11”, “1111-111”, or “11111111111”, which means a regex like [0-9-]+ would be insufficient. Instead, we need to a regex like this: [0-9]{3}-[0-9]{4}: precisely three digits, then a dash, then precisely four digits.
- You can also use braces to specify ranges, either bounded or unbounded. For example, [a-z] {1,3} means “match one, two, or three lowercase letters”, and [a-z]{3,} means “match at least three, but potentially any number more.”
- Using this technique, you should be able to see that the regexes colou?r and colou{0,1} r are identical – both match either zero or one “u” in the word “colour”.

Meta Characters

- There are several characters that regexes give special meaning, and at least three of them are used extensively.
- First up, the most used, most overused, and most abused of these is the . character – a period – which will match any single character except a line break. So, the regex c.t will match “cat” but not “cart”. If you use . with the * quantifier it means “match one or more of anything that isn’t a line break,” which is probably the most common regex you’ll come across.
- The reason for its use should be easy to recognize: you don’t need to worry about crafting a specific regex, because .* will match almost everything. The problem is, being specific is sort of the point of regular expressions: you can search for precise variations of text in order to apply some processing – too many people rely on .* as a crutch, without realizing it can introduce subtle mistakes into their expressions.
- As an example, consider the regex we wrote to match phone numbers like 555-5555: [0-9] {3}-[0-9]{4}. You might think “maybe some people will write “555 5555” or “5555555”, and try to make your regex looser by using .* instead, like this: [0-9]{3}.*[0-9]{4}.
- But now you have a problem: that will match “123-4567”, “123-4567890”, and even “123-456-789012345”. In the first instance, the .* will match “-“; in the second, it will match “-456“; and in the third it will match “-456-78901” – it will take as much as needed in order for the [0-9]{3} and [0-9]{4} matches to work.
- Instead, you can use character classes with quantifiers, for example [0-9]{3}[-]*[0-9] {4} means “find three digits, then zero or more spaces and dashes, then four digits.” or negated character classes. You can also use negated character classes to match anything that isn’t a digit, so [0-9]{3}[^0-9]+[0-9]{4} will match a space, a dash, a slash, and more – but it won’t match numbers.

Anchors

- There are two meta characters used to describe the beginning and end of each line: ^ and $ respectively.
- These are called anchors, because they are useful to restrict the kind of matches you're looking for – you literally anchor the match to one or both parts of the input line.
- Now, there's a small complexity here: regexes were originally designed to handle one line of text at a time, but nowadays it's much more common to parse hundreds or even thousands at a time. To preserve backwards compatibility, most programmatic regex engines (i.e., ones you use in code) consider the start and end of the line to be the start and end of your whole text – no matter how many line breaks it has, and you need to enable a special multi-line mode. Regex engines built into text editors tend to enable multi-line by default, so this is a non-issue.
- You've already seen the "g" (global) regex modifier that matches all results rather than just the first, and also the "i" (insensitive) modifier that ignores letter case. Well, there's also an "m" modifier that enables multi-line mode.
- In some languages there are flags to accomplish the same thing,
    - e.g. in Swift you use .AnchorsMatchLines,
    - in Objective-C you use NSRegularExpressionAnchorsMatchLines,
    - in Java you use Pattern.MULTILINE,
    - in C# you use RegexOptions.Multiline, and so on.

Meta Sequences

- Regular expressions have a handful of special characters that can be used in place for character classes. These meta sequences come in pairs distinguished by letter casing, for example: \w (a lowercase W) means “any word character” is equal to the character class [A-Za-z0-9_], and \W (a capital W) means “the opposite of any word character”. This means [A-Za- z0-9_]* and \w* are identical.
- Similarly, you have \d to match any digit and \D to match any non-digit, as well as \s to match any whitespace and \S to match any non-whitespace. The whitespace sequence is equivalent to combining all the whitespace meta characters that are used in most programming languages, including \t for “tab”, \n for line break, plus a space.
- use these meta sequences to write regular expressions that match the following:
    - 1. Variables used in programming.
        - \w* does exactly the same thing and is significantly shorter.
    - 2. A phone number in the format 111-1111.
        - could be solved using our previous phone number regex, [0-9]{3}-[0-9]{4}, but now we can use \d to mean “any digit,” giving \d{3}-\d{4}
    - 3. Names that follow the format “Paul Hudson” and “Taylor Swift”.
        - this does not need meta sequences, so you should have a regex similar to [A- Z][a-z]+ [A-Z][a-z]+.
    - 4. A book ISBN in the format 1-111111-11-1, where the dashes may or may not be present.
        - You should have used the \d meta sequence again, along with the brace quantifiers and question marks to handle zero or one dashes. Putting that all together, the regex should be something like this: \d{1}-?\d{6}-?\d{2}-?\d{1}.
- See what I mean about more advanced regexes starting to look like line noise?

Looking For Words

- Regexes have two particularly interesting meta sequences in the form of \b and \B. The former matches word boundaries, and the latter matches non-word boundaries, but the former is far more common.
- An initial attempt might be as simple as \w+, meaning “match one or more words. However, this has a problem: that meta sequence evaluates to [A-Za-z0-9_], which doesn’t include apostrophes. So, it will match “doesn” as one word and “t” as another, rather than matching “doesn’t”.
- A slightly smarter guess might be [\w']+, meaning “any word character or apostrophe, repeated one or more times.” This works better, and in our test string will correctly match almost all the words. What it doesn’t match is the word “no-one”, which is hyphenated here although it can be written in several different ways. This is also not considered a word character, so the current regex matches “no” and “one” rather than “no-one”. To fix that, we need to add hyphens to the regex pattern, like this: [\w'-]+.
- However, what if the user has entered curly quotes – most coders are used to writing ' for apostrophes, but really it ought to be ’, which is a subtle difference but commonly used thanks to “smart quotes” features in word processors. In that case, your regex would need to be [\w'’-]+. But then what if the user entered numbers with commas, like “1,000,000”? That should also be matched as a single word, so you would need to update your regex again, and sooner or later you might decide this is more trouble than it’s worth.
- Let’s pursue a different option: character set negation. Try entering the regex [^]+ and you’ll see it matches everything that isn’t a space. This is pretty close – even correctly having “doesn’t” as a single match – but it also screws up the punctuation, because it matches “toast.” rather than “toast”, and “now,” rather than “now”.
- This is where the word boundary meta sequence can be useful: punctuation is considered to mark the boundaries of words, so we can write our regex so that it matches non-space characters that end with a word boundary of any kind.
- Try this regex now: [^]+\b.
    - Straight quotes or curly quotes, hyphens, commas, and more are now all correctly matched, and the best bit is that it’s easy to read – the holy grail of regular expressions.

Grouping

- Consider this problem: we want to match dates like “April 25th”, but we want to be generous enough to allow them to be written like “Apr 25” instead. Can you write a regex to do just that?
- With that kind of optionality, you might have written something like this:
  - Apri?l? \d\dt?h?
- That does indeed match “April 25th” and “Apr 25”, but it also accepts “Aprl 25h”, which is gibberish.
- A solution is to use grouping, which lets you make sub-expressions inside your regex, then apply a quantifier. To do this, use parentheses. For example, the “il” in April and the “th” in “25th” should either be present in their entirety or not present at all – zero or one matches of that particular text. We write that as (il)? and (th)?, so we can craft a much better regex like this:
  - Apr(il)? \d\d(th)?
- If you use that regex with the test string “Apr 25th” you’ll notice that Regex101 colors “April 25” in one color, and “th” in another. This isn’t a mistake in our regex, in fact it’s a feature of regexes that we’ll get to soon: each time you use a group like this, it gets matched individually so that you can manipulate it more easily.
- Without groups, you can test “did the whole string match my regex?” which is often useful, but with groups you’ll be given an array of all the matched values from the groups you created, so you can say “what value did group 1 contain?” To see this in action, enter “April 25th” as the test string and look on the right side of Regex101: you’ll see a box marked “Match Information”, showing “il” and “th” – the matches for our two groups.
- Note: if you use a group without a quantifier, it must be matched exactly once. So, April and Apr(il) both will only match the test string “April”.

Grouped alternatives

- Groups become really powerful when you use them to specify multiple variations. This is done by writing a pipe symbol, |, inside your groups, like this: (foo|bar). That will match either word but not both, so you can allow variation from users within the same regex.
- EXAMPLE
    - The situation is this: you want to parse a sentence of user text that matches the structure “I like paintings by Caravaggio” or “I like sculptures by Rodin”. You don’t know who the artist might be, but you do know that it’s going to be a painting or a sculpture – we don’t want to match “I like music by Sam Smith”.
    - Let’s start with a regex that matches “I like paintings by...” followed by any name:
        - I like paintings by \w+
    - That matches artists with a single surname, as both our examples have. If you didn’t know how many to match, you could use .*.
    - What we have so far will match “I like paintings by Caravaggio” and “I like paintings by Rodin”, but not “I like sculptures by Rodin” – to do that we need a grouped alternative for “paintings”, like this:
        - I like (paintings|sculptures) by \w+
    - And that’s it: we can now throw any painters or sculptors at that regex, and it will work fine.

Greedy, lazy, and eager matching

- At this point you know the most important ways of building regular expressions, but there are few bonus techniques you should learn if you want to feel really in control.
- The first of these is understanding eager matching. Regex engines are considered “eager” because they are programmed to return the first valid match, even if there is a better match available.
- This is important when working with grouped alternatives, because the regex engine checks them in the order you provide. For example, considered the group (get|getName|set| setName): when given the string “setName”, which of those four options do you think will be matched?
- The answer is “set”, even though “setName” would be a better match given the list of options and the test string being used. This happens because regular expressions are eager: they go linearly through the list of possibilities you’re looking to match, and use whichever one works first. This makes them efficient, but also forces you to consider your expression carefully: the group (getName|get|setName|set) does the same thing as (get|getName|set| setName) but will always match the longest string it can.
- As well as being eager, the three quantifiers we have used, *, +, and ? are also greedy by default, which means they will match as much as possible. I already explained that it’s preferable to avoid .* if possible, but if you understand greediness you can make .* more useful.
- Consider this text: "That's it: I'm invoking Space Corp Directive 68250." said Rimmer. "68250?" replied Kryten, "but sir surely that's impossible without at least one live chicken and a rabbi?"
- That’s a complicated piece of text, with multiple types of punctuation: a colon, a couple of periods and question marks, apostrophes, quote marks, and commas.
- What kind of regex would you write if you wanted to match each piece of speech?
- Your first attempt might be something easy like this: ".*". That is, “a quote mark, then anything, then another quote mark.” The problem is that * is greedy by default, which means it will try to match as much text as possible before it returns.
- I chose the current test string because it’s the perfect demonstration of greediness: it starts with a quote mark and ends with a quote mark, which means .* will return the entire string as a single match even though there are multiple quote marks that could be matched sooner.
- That’s greediness: * will take as much as it can – think of * as meaning “match this thing as many times as possible.” The alternative is called lazy matching and means “match this thing as few times as possible,” and it’s enabled by adding a question mark to your existing quantifier.
- In our current example that means using the regex ".*?", which means “match a quote, then as few characters as possible before we reach the next quote.”

- As I have now said a couple of times, using .* is vague and there are more precise alternatives. In this case, we could also use "[^"]+" to get the same result as a lazy * – it means "match a quote, then match everything up to the next quote."

Escape characters

- Imagine a to do list file that was structured something like this:
    - 1. Feed cat
    - 2. Organize sock drawer
    - 3. Take over world
- You could write something like this: → \d+. [\w]+
- And while that's a valid solution, it might not work for the reason you think it works. Remember, . is actually a meta character that means "anything at all." It doesn't have the * quantifier here so it will only match a single character, but in this case it means strings like the below will match:
    - 11 Take over the world
- It matches because the first 1 matches the \d, and the second 1 matches the . character. What we wanted the regex to mean was "one or more digits, then exactly one period, then exactly one space, then any number of word characters or spaces."
- To make . mean a period – and not be a meta character that matches anything – we need to escape it. This will be a familiar concept to you when working with strings: if you write a string like "Hello" and need to include quotes inside it, you escape the quotes by preceding them with a backslash: "\"Hello\" she said". The backslash means "this is a literal quote, and not a quote that ends the string." This same approach is used with regular expressions: we use \. to mean "a literal period, not the 'any character' meta character."
- we use \. to mean "a literal period, not the 'any character' meta character."
- So, if we want our regex to explicitly match a period, it needs to look like this:
    - \d+\. [\w]+
- Escaping is complicated for two reasons. First, the rules for escaping special characters are different depending on whether you're inside a character class or not. If you are inside a character class, you need to escape the following characters if you want to match them literally:], \, ^, -. This is because they have special significance inside character classes. Outside of character classes, you need to escape lots more things: ., ^, $, *, +, ?, (,), [, {, \, and | all must be escaped, because otherwise they have special meanings.
- Note: If you're using a language that wraps regexes in delimiters – JavaScript and PHP, for example – you will also need to escape that delimiter. The most common delimiter is /, so you will need to write \/. Regex101 uses / for a delimiter, so this warning applies there too: write \/ on Regex101!
- The second reason escaping is complicated – and trust me, this bit hurts my brain – is when you use regexes inside programming strings. If you're solely using them inside a text editor then you have no problem, but when you use them inside a programming

language then you need to double escape things. Let me demonstrate with some pseudocode:
  - myString = "\d+\. [\w]+"
- Because most programming languages use \ as their own escape sequence, there's a problem with that pseudocode: each of those backslashes will be interpreted as an escape sequence in your programming language, rather than than an escape sequence for the regex.
- The solution, like I said, is to double escape. So, whenever you write \ you need to write \\, making the above pseudocode into this:
  - myString = "\\d+\\. [\\w]+"
- When \\. is read by whatever compiler you use, it will be interpreted as “a literal backslash followed by a period,” i.e. the \. character. That then goes to the regex engine, which interprets it as “a literal period, rather than a meta character.”
- Now, brace yourself, because this next bit might hurt. If you're actually trying to match a backslash, you need to escape it in your regex no matter what. So, if you're using a text editor for your regex, it will be \\ rather than just \. If you're using a programming language, however, you need to double escape, which means that if you want to match a single backslash you need to write this:
  - myString = "\\\\"
  - Yes: four backslashes are required in order to match a single backslash. Sorry!

Back references and lookahead

- There are two advanced techniques I do want to demonstrate, but I must stress they are infrequently used.
    - The first is called back references, and it allows you to make reference to a previous match.
    - The second is called lookahead, and it allows you to make a regex match only if it's followed by something else.
    - Note: Both of these two features are advanced for computers as well as humans. They are computationally expensive, so if you intend to add them into your programs do so very carefully indeed.
- Let's start with back references. These are an extension of regex groups, and let you refer to a previously matched group later on in a regex. To give you an example, let's invent our own Markdown-like syntax for adding formatting to text:
    - *Text written like this will be bold*
    - /Text written like this will be italic/
    - _Text written like this will be underlined_
    - *Text written like this is just text_
- The first three lines start with a symbol and end with the same symbol, but the fourth one does not and so is ignored.
- A naïve regex might look like this:
    - [*\/_].*?[*\/_]
- That is, "any of *, /, or _, followed by anything at all (in lazy mode), followed by another *, /, or _" The problem is that it will match all four lines, including the one where the start symbol differs from the end symbol
- What we need to do is write a regex where the second symbol must be the same as whatever was matched by the first group, and that means using back references. Whenever you create a capture group using parentheses, you can refer to whatever that group matched elsewhere in your regex using numbers counting from 1. For example, the first match group is referred to as \1.
- Using this technique, we can write a new regex that only handles correct formatting:
    - ([*\/_]).*?\1
- That means "match either *, /, or _ as group one, then anything, then another of the same character that was matched in group one."
- EXAMPLE
    - First, write a regex that finds all the letters that are doubled in the string "The happy aardvark went swimming in the pool."
        - one single letter is represented as \w, so to find double letters we need to wrap that in a group to make (\w) then add a back reference to that group, making (\w)\1.
    - Second, write a regex that finds the doubled word in the string "Then the theater manager kicked the the back of the seat."

    - You might have tried to craft a regex like (\w+) \1, to mean “any number of word characters, followed by a space, followed by the same sequence of word characters.” While that would correctly match “the the” after “kicked”, it would also – incorrectly! – match “the the”, where the second “the” was part of “theater”. This is not a repeated word.
    - The correct solution is to use the word break meta character, \b, to ensure that a word ends immediately before and after the match. So, you’d write this: \b(\w+) \1\b. That means “a word break, then a word, then a space, then the same word again, the another word break.” Word breaks are clever enough to be matched against full stops, beginnings and ends of a string, and more.
- So, that’s back references: the ability for one part of your regex to refer to what was already matched in an earlier part of your regex.

Positive and negative lookahead

- As the name suggests, lookahead works by looking beyond your match group to ensure that something does or does not follow it. These two forms are called positive and negative lookahead, and both can help you solve difficult problems.
- Let's work with another real example: put this text into the test string of Regex101:
    - Alabama, Alaska, Arkansas, California, Colorado, Connecticut, Delaware, Florida, Georgia, Hawaii
- That's a list of some US states, separated by a comma. How would you write a regex to match all the states that end with "a"? A first attempt might look something like this:
    - \w+a,
- That means "one or more word characters, then an 'a', then a comma." That's quite an easy regex compared to some of the others we've seen, but it has a problem: it matches "Alabama,", "Alaska,", "California," and so on – the comma is included with the matching text. That's unpleasant, and would require further processing to be useful – at least, that is, if positive lookahead didn't come to the rescue.
- Positive lookahead lets us write a regex that means "find any number of letter characters, then an 'a', but only if that 'a' is immediately followed by a comma" – and it doesn't include the comma in the match. The regex engine will match the text we ask from it, then peek a little bit beyond to make sure our lookahead is correct.
- Positive lookahead matches look like regular groups, but they start with ?=. For example, to look ahead for the existence of a comma, we would write (?=,) immediately after the "a", which makes the whole regex this:
    - \w+a(?=,)
- Negative lookahead flips things around: it allows you to check for the absence of something after your match. It's written like ?!, but otherwise works like positive lookahead. Let's start with a slightly silly example – here's a list of statements:
    - Sam Smith is alive
    - Michael Jackson is no more
    - Taylor Swift is alive
    - John Lennon is dead as a dodo
    - Adele Adkins is alive
    - Elvis Presley is pushing up the daisies
- So, each singer is either "alive" or some form of colloquialism for dead. Negative lookahead lets us write a regex that matches the names of the dead people, even though we don't know how they are described – except that they are not described as "alive".
- To write this regex, we need to first match a first name and last name, which can be done using \w+ \w+. We then need to match the exact string " is ", and finally the negative lookahead: we want this line to match only if the next word is not "alive".
- So, the total regex is this:
    - \w+ \w+ is (?!alive)
    - That will match only lines that have dead singers.

Match Groups

- We've looked at groups a little so far, but I want to go into more depth now. Let's start with the same regex from the end of the previous chapter: \w+ \w+ is (?!alive), which means "a word followed by another word, but only if it isn't followed by 'is alive'." We can give that regex a test string like the below, and it will correctly tell us that lines 2, 4, and 6 match:
    - Sam Smith is alive
    - Michael Jackson is no more
    - Taylor Swift is alive
    - John Lennon is dead as a dodo
    - Adele Adkins is alive
    - Elvis Presley is pushing up the daisies
- In regexes this is done using groups. We already had a simple example of this with the regex Apr(il)? \d\d(th)?, where "il" and "th" become match groups if they are present in the test string. So, rather than just saying "that date matched our regex", we can see for sure whether the use entered "Apr" or "April", and "25th" or "25".
- Let's give it a try now – enter the test string from above, plus the regex \w+ \w+ is (?!alive). You should see three matching lines, but notice 1) it matches the name plus "is" and no further, and 2) nothing appears in the Match Information box.
- The first of the problems is the result of our regex: the lookahead part is not attached to the matches we're trying to make. It's a way for us to say "match A only if B does or does not follow," and not "match A and B." In order to fix this, we need to add some sort of text matching after the lookahead, so that we can say "we don't want B to follow A, but we still want to match what was in B."
- So, our current regex is this:
    - \w+ \w+ is (?!alive)
    - We're going to modify it to this:
    - \w+ \w+ is (?!alive).*
- With that tiny change – just adding .* to the end of the regex – all of the matching lines will be colored in Regex101, reflecting the fact that the whole line is now matched by the regex. However, practically not a lot has changed: we're still matching whole lines.
- Enter groups: every time you use a group, i.e. wrapping part of your regex in parentheses, you create match groups. These are a way of saying to the regex engine, "I want to know that this line matched, but I want to know exactly what text was matched." In our current example, we might want to know the name of each singer, or what euphemism was used to describe them being an ex-parrot. To do that, all we have to add is parentheses around the bits we care about, like this:
    - (\w+ \w+) is (?!alive)(.*)
- Now look in the Match Information box, and you'll see what those match groups have done for us: you'll see Match 1, Match 2, and Match 3 all listed, with each one having two matches inside it. For example, Match 1 contains "Michael Jackson" and "no more", and Match 2 contains "John Lennon" and "dead as a dodo".

Non-matching groups

- So far you've learned that creating groups is how you accomplish three separate tasks:
  - 1. Add quantifiers to a string rather than a character, e.g. Nov(ember)?
  - 2. Specify alternate text, e.g. (paintings|sculptures)
  - 3. Retrieve specific parts of the match for further processing.
- That last one is particularly powerful when combined with replacements, which is covered in the next chapter. However, the other two create match groups even though you might not need them – if all you care about is that the user specified the 11th month, it doesn't matter whether they wrote "Nov" or "November". This means you get match information for things you care about ("what was the name of the singer?") and things you might not ("what it Nov or November?") mixed together, which clutters up your match information.
- Regexes have a solution for this in the form of non-matching groups, often call non-capturing groups. To make a regular group you just surround part of your regex in parentheses, for example (movie|film) will match either "movie" or film". To make that a non-matching group – a group that doesn't generate any match information – add ?: just inside the opening parenthesis,
  - like this: (?:movie|film).
- That is: we expect to receive the exact text "My favorite" followed by either movie or film, followed by the exact word "is", followed by any text. I've made the movie/film group non-matching because we don't care what the user said, but I made the movie name a group because we do care what that is.
  - With that regex in place, try adding this test string:
    - My favorite movie is Withnail and I.
    - My favorite film is Life of Brian
  - You'll see it has two items in the Match Information box: "Withnail and I" and "Life of Brian". Now try changing (?:movie|film) to (movie|film) and you'll see the difference: the regex now separately captures whether "movie" or "film" was used.
- Non-matching groups use similar syntax to lookahead so I want to present it all together so you can see the distinction more clearly:
  - Nov(?=,) is positive lookahead: I want my regex to be followed by a comma.
  - Nov(?!,) is positive lookahead: I don't want my regex to be followed by a comma.
  - Nov(ember)? is a matching group that may or may not appear: I want the user to be able to write "Nov" or "November".
  - Nov(?:ember)? is a non-matching group that may or may not appear: I want the user to be able to write "Nov" or "November", but I don't want the "ember" part as a match group.
- It's good practice to use non-matching groups unless you specifically want the text inside a group, because it stops your match results filling up with needless information that you need to skip over in order to find the important stuff.

- Time to try it for yourself. Imagine users describing the design of an app by typing out how it should look. You've been given the task to write four of the regexes to recognize various parts of the app, specifically:
    - 1. Given the text The app has 3 tabs you need to pull out the number 3. Users might also write The app has 1 tab rather than 1 tabs.
        - To handle the variation between "tab" and "tabs" means using the regex tabs?, so that's one problem down. Capturing the "3" from "3 tabs" is a matter of matching one or more digits using \d+ then placing that inside a group to ensure it's captured. So, the final regex is The app has (\d+) tabs?.
    - 2. Given the text The app has the icon "appIcon.png", you need to pull out "appIcon.png".
        - The app has the icon "([^"]+)". Again, a capturing group is required to pull out the icon name, but I've used an inverse set rather than .* to make it a bit more precise.
    - 3. Given the the text row 4 has a red background you need to pull out "4" and "red". Users might also write row 1 has an orange background – the number varies, and if the color starts with a vowel than "an" will be used rather than "a".
        - We'll need two capturing groups this time, plus the optional "n" to handle "a" and "an". So, the final regex is row (\d+) has an? (\w+) background.
    - 4. Given the text the right bar button has the title "Map", you need to pull out "right" and "Map". Users might also specify a "left", "center", or "centre" bar button, and you should catch them all.
        - To ensure the user provides one of several options we need to use a grouped alternative, like this: (left|right|center|centre). It needs to be a regular matched group because we want to know which one the user chose. As for the bar button title, that's just another inverse set inside quote marks, i.e. "([^"]+)".
        - So, the total regex is this:
            - the (left|right|center|centre) bar button has the title "([^"]+)"

Replacements

- So far we've been working exclusively with text matching, and you already know enough to be able to parse all sorts of natural language using regular expressions. This alone is powerful: you can now write code to parse a million lines of text in a couple of seconds, which is incredible.
- But there's more: regular expressions are also useful for replacements. This means you can match text in one format using all the techniques you already learned, then rewrite it in a new format with the same incredible efficiency. Even though regex replacements are powerful, I left them here at the end of the chapter because they build on other concepts you needed to learn first.
- In fact, at this point, you're going to find replacements a cinch: they use the same search syntax for regular expressions, and either use literal strings for replacements or parts of the match.
- Let's look at literal string replacement first, because that's most easy. To do this, I want you to start by reloading the Regex101 page in your browser so that it's completely reset – the replacement box can be a bit glitchy if you've been using the page previously.
- With the page reloaded, I want you to click at the bottom where it says "Substitution". All being well, that should make two new form fields appear: one text entry box saying "substitution", and a large text area beneath it that's empty. If it doesn't work, reload the page and try it again.
- When you perform regex replacement, you get to specify what should be put in place of the whole matching regex. With literal string replacement, that means you're replacing whatever matches the complete regex with a simple piece of text. Imagine your documentation team had been busy writing thousands of pages of text describing how to use the latest, greatest software your team had built. But there's a problem: sometimes they said to click the center button, and other times they said to click the centre button – they mixed their spellings. After a long fight, no one could agree which spelling was the correct one, so your boss makes a decision: replace both "center" and "centre" with the word "middle" to keep everyone happy.
    - This is an intentionally trivial example, but it's a safe place to start. First, put this regex into Regex101: \b(center|centre)\b. Now add this text string:
        - The button should be aligned to the center of the screen
        - The button should be aligned to the centre of the screen
    - Finally, in the top substitution box, write "middle". You should see the bottom substitution box update with this:
        - The button should be aligned to the middle of the screen
        - The button should be aligned to the middle of the screen
    - Notice how "middle" is used to replace only the parts of the test string that match our regex - everything else is completely ignored. This makes regex replacement really non-invasive.

Replacing with matched groups

- Where regex replacement's real power lies is in its ability to replace text by drawing on match groups. Every time you create a match group using parentheses, the contents of that match are available to you as a special value named using a dollar sign then the group number.
  - So, $1 is the first match group,
  - $2 is the second, and so on.
  - You can also use $0 to mean "the whole matched regex."
- Let's start with $0 for now. The situation is this: a writing team has crafted lots of great content for you new website, but sadly they aren't very good at HTML. So, instead of creating proper HTML hyperlinks so they are clickable, they've just typed out plain text links. So, your source text is this:
  - There are lots of great websites, but https://gum.co/proswift is probably my favorite. A second favorite is https://gum.co/ objcswift – that's worth reading too.
- Using regular expressions, we can parse that text, find all web links, then convert them into real HTML hyperlinks using the $0 replacement match.
- Enter the above test string into Regex101, then add this regex:
  - https?:\/\/[^]+
- That means "the exact text 'http', follow by zero or one 's', then '://', then anything that isn't a space." There's no group there because we want to use the entire contents of the regex, so to create the correct output you need to enter this in the substitution box:
  - <a href="$0">$0</a>
- Remember, $0 means "use everything that matched the regex", so that will produce a hyperlink pointing to the URL, then use the same text for the clickable text. The output HTML is this:
  - There are lots of great websites, but <a href="https://gum.co/proswift">https://gum.co/proswift</a> is probably my favorite. A second favorite is <a href="https://gum.co/objcswift">https://gum.co/objcswift</a> – that's worth reading too.
- Creating replacements using your match groups is done using $1, $2, and so on. To demonstrate this, I want to give you a slice of text taken from Monty Python and the Holy Grail – put this into the sample text box:
  - King Arthur: Go and tell your master that we have been charged by God with a sacred quest. If he will give us food and shelter for the night, he can join us in our quest for the Holy Grail.
  - French Soldier: Well, I'll ask him, but I don't think he will be very keen. Uh, he's already got one, you see.
  - King Arthur: What?
  - Sir Galahad: He said they've already got one!
  - King Arthur: Are you sure he's got one?
  - French Soldier: Oh yes, it's very nice!

- There are six lines in total there, each following a specific format: the name of the speaker, a colon, then their lines. We're going to convert that plain text into HTML by using regex replacement, but we're going to use match groups to add some styling. Specifically:
    - • Every line should start with a <p> tag.
    - • The speaker names should be wrapped inside <strong> to help it stand out.
    - • We'll retain the colon between speaker name and line text.
    - • We're going to place the line text inside quote marks.
    - • Every line should end with a </p>.
- Because we need to create groups from the speaker name and their line, the regex is simple enough:
    - (.*): (.*)
- That creates two match groups, so we can refer to $1 and $2 to place the speaker name and their line wherever we want them. Put this text into the substitution box:
    - <p><strong>$1</strong>: "$2"</p>
- With that in place, the output is now beautiful HTML:

Back To Grep

- We covered the grep command in the terminal chapter, but that was before I had introduced you to the whole world of regular expressions and so it's worth revisiting now.
- If you recall, grep is the command that searches inside files for matching text, but that text doesn't need to be fixed – it can be a regular expression. This means everything you have learned about regular expressions can now be applied to grep on the command line: you can search through huge amounts of text for any regex you please, then pipe the results from that search into other terminal commands.
- To give you a practical example of this, every time a web server serves up a page, it adds a line like this to the server log:
    - 125.130.217.146 - - [21/Apr/2016:06:27:50 +0000] "GET /buy.html HTTP/1.1" 200 379 "https://www.awesomeinc.com/index.html" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/ 537.36 (KHTML, like Gecko) Chrome/51.0.2704.63 Safari/537.36"
- There's a lot of information in that single line, and it's beyond the remit of this book to describe what it all means. However, the important bit is this: GET /buy.html – that's the bit that means a user somewhere in the world requested messages.html.
- Now, your company is busy doing some testing to try to improve sales: if they change the text of the "buy" button does it make sales higher or lower? To find out, they wrote some simple code to display eight different pieces of buy text, each linking to the same buy.html file but providing a parameter in the URL: "buy.html?type=1", "buy.html?type=2", and so on.
- Those parameters are ignored by the web server but get stored in the log, which means after 30 days have passed you have 1,000,000 lines in your web server log, of which about 10,000 contain a reference to "buy.html". Management want to know which buy button was clicked the most, which means you need to find all the references to "buy.html" and count which "type" parameter was used most.
- There are a couple of parts to this task, and it's been a little while since we looked at terminal programs, so let's walk through this slowly.
- First, the regex would look something like this:
    - buy.html\?type=\d
- I've escaped the question mark because we want a literal question mark rather than meaning a zero-or-one quantifier. However, this isn't needed when using grep: unless you specify otherwise, grep uses a tiny subset of regular expressions, so things like +, ?, and | won't do anything, and instead just represent their literal characters.
- As a result of this, the regex for grep is even simpler:
    - buy.html?type=\d
- That will match all lines where there's a buy link, but our job is to count the matches to see which button type did best. To make that work, we're going to invoke grep with its -o option. This is new, but I didn't introduce it earlier because it doesn't much sense without regular expressions at hand! What it means is "output only the part of the line that

matches my search." So, when you have a long server log line that has all the other information you saw earlier, enabling -o means "just print out the "buy.html?type=4" part.

- So far our command looks like this:
  - grep -o "buy.html?type=\d" server.log
- That will print out 10,000 lines of "buy.html?type=1", "buy.html?type=5" and so on. The next step is to count the various instances to see how often they appear, and to do that we can use the -c parameter to uniq that we learned previously. Remember, though: uniq expects identical lines to be adjacent to each other, so we need to pass the output through sort first.
- That makes our final command line look like this:
  - grep -o "buy.html?type=\d" server.log | sort | uniq -c
- The output – if server.log actually existed, and actually had 10,000 rows of matching data – would look like this:
  - 4601 buy.html?type=8
  - 1891 buy.html?type=1
  - 955 buy.html?type=4
  - 602 buy.html?type=3
  - 598 buy.html?type=2
  - 540 buy.html?type=6
  - 499 buy.html?type=5
  - 314 buy.html?type=7
- Boom: you can see at a glance that type 8 was by far the best, followed by type 1, then type 4.
- This is the beautiful crossroads where regex and terminal commands combine to do marvelous things just with a few seconds of typing.

Full-power regex with egrep

- By default, grep disables most regular expression syntax: * works to mean "zero or more", and you already saw that \d works to mean "any digit", but the characters ?, +, {, |, (, and) lose their special meanings.
- Don't worry: if you have a complex regex that you want to use with grep, just call it with the -E option to enable extended regular expressions. This usage is so common that Unix systems create a special grep alias called egrep that is identical to calling grep -E.
- So: regular grep supports basic regexes with character classes, meta sequences, and *, but egrep turns on the full range of power so you can search using any regex you can think of.