Regular Expressions

Regular expressions are extremely useful in extracting information from text such as code, log files, spreadsheets, or even documents. They can do text manipulation much, much faster than you'll be able to using other strings tools such as compare, contains, replace, etc. A well-crafted regex can perform your find, compare and replace all in one action

The first thing to recognize when using regular expressions is that everything is essentially a character, and we are writing patterns to match a specific sequence of characters (also known as a string). Most patterns use normal ASCII, which includes letters, digits, punctuation and other symbols on your keyboard like %#$@!, but unicode characters can also be used to match any type of international text.

And so characters include normal letters, but digits as well. In fact, numbers 0-9 are also just characters and if you look at an ASCII table, they are listed sequentially. In this case, the character \d can be used in place of any digit from 0 to 9. The preceding slash distinguishes it from the simple d character and indicates that it is a metacharacter. \D does the opposite - it matches and non-digit character

RegEx has the concept of a wildcard, which is represented by the . (dot) metacharacter, and can match any single character (letter, digit, whitespace, everything). You may notice that this actually overrides the matching of the period character, so in order to specifically match a period, you need to escape the dot by using a slash \. accordingly.

There is a method for matching specific characters using regular expressions, by defining them inside square brackets. For example, the pattern [abc] will only match a single a, b, or c letter and nothing else.

In some cases, we might know that there are specific characters that we don't want to match too, for example, we might only want to match phone numbers that are not from the area code 650.

To represent this, we use a similar expression that excludes specific characters using the square brackets and the ^ (hat). For example, the pattern [^abc] will match any single character except for the letters a, b, or c.

But what if we want to match a character that can be in a sequential range characters? Do we have no choice but to list them all out?

Luckily, when using the square bracket notation, there is a shorthand for matching a character in list of sequential characters by using the dash to indicate a character range. For example, the pattern [0-6] will only match any single digit character from zero to six, and nothing else. And likewise, [^n-p] will only match any single character except for letters n to p.

Multiple character ranges can also be used in the same set of brackets, along with individual characters. An example of this is the alphanumeric \w metacharacter which is equivalent to the character range [A-Za-z0-9_] and often used to match characters in English text. The corresponding \W character will match anything that is not in that range.

We've so far learned how to specify the range of characters we want to match, but how about the number of repetitions of characters that we want to match? One way that we can do this is to explicitly spell out exactly how many characters we want, eg. \d\d\d which would match exactly three digits.

A more convenient way is to specify how many repetitions of each character we want using the curly braces notation. For example, a{3} will match the a character exactly three times. Certain regular expression engines will even allow you to specify a range for this repetition such that a{1,3} will match the a character no more than 3 times, but no less than once for example.

This quantifier can be used with any character, or special metacharacters, for example w{3} (three w's), [wxy]{5} (five characters, each of which can be a w, x, or y) and .{2,6} (between two and six of any character).

A powerful concept in regular expressions is the ability to match an arbitrary number of characters. For example, imagine that you wrote a form that has a donation field that takes a numerical value in dollars. A wealthy user may drop by and want to donate $25,000, while a normal user may want to donate $25.

One way to express such a pattern would be to use what is known as the Kleene Star and the Kleene Plus, which essentially represents either 0 or more or 1 or more of the character that it follows (it always follows a character or group). For example, to match the donations above, we can use the pattern \d* to match any number of digits, but a tighter regular expression would be \d+ which ensures that the input string has at least one digit.

These quantifiers can be used with any character or special metacharacters, for example a+ (one or more a's), [abc]+ (one or more of any a, b, or c character) and .* (zero or more of any character).

Another quantifier that is really common when matching and extracting text is the ? (question mark) metacharacter which denotes optionality. This metacharacter allows you to match either zero or one of the preceding character or group. For example, the pattern ab?c will match either the strings "abc" or "ac" because the b is considered optional.

Similar to the dot metacharacter, the question mark is a special character and you will have to escape it using a slash \? to match a plain question mark character in a string.

When dealing with real-world input, such as log files and even user input, it's difficult not to encounter whitespace. We use it to format pieces of information to make it easier to read and scan visually, and a single space can put a wrench into the simplest regular expression.

The most common forms of whitespace you will use with regular expressions are the space (), the tab (\t), the new line (\n) and the carriage return (\r) (useful in Windows environments), and these special characters match each of their respective whitespaces. In addition, a whitespace special character \s will match any of the specific whitespaces above and is extremely useful when dealing with raw input text.

So far, we've been writing regular expressions that partially match pieces across all the text. Sometimes this isn't desirable, imagine for example we wanted to match the word "success" in a log file. We certainly don't want that pattern to match a line that says "Error: unsuccessful operation"!

That is why it is often best practice to write as specific regular expressions as possible to ensure that we don't get false positives when matching against real world text.

One way to tighten our patterns is to define a pattern that describes both the start and the end of the line using the special ^ (hat) and $ (dollar sign) metacharacters. In the example above, we can use the pattern ^success to match only a line that begins with the word "success", but not the line "Error: unsuccessful operation". And if you combine both the hat and the dollar sign, you create a pattern that matches the whole line completely at the beginning and end.

Note that this is different than the hat used inside a set of bracket [^...] for excluding characters, which can be confusing when reading regular expressions.

Regular expressions allow us to not just match text but also to extract information for further processing. This is done by defining groups of characters and capturing them using the special parentheses (and) metacharacters. Any subpattern inside a pair of parentheses will be captured as a group. In practice, this can be used to extract information like phone numbers or emails from all sorts of data.

Imagine for example that you had a command line tool to list all the image files you have in the cloud. You could then use a pattern such as ^(IMG\d+\.png)$ to capture and extract the full filename, but if you only wanted to capture the filename without the extension, you could use the pattern ^(IMG\d+)\.png$ which only captures the part before the period.

When you are working with complex data, you can easily find yourself having to extract multiple layers of information, which can result in nested groups. Generally, the results of the captured groups are in the order in which they are defined (in order by open parenthesis).

Take the example from the previous lesson, of capturing the filenames of all the image files you have in a list. If each of these image files had a sequential picture number in the filename, you could extract both the filename and the picture number using the same pattern by writing an expression like ^(IMG(\d+))\.png$ (using a nested parenthesis to capture the digits).

The nested groups are read from left to right in the pattern, with the first capture group being the contents of the first parentheses group, etc.

As we mentioned before, it's always good to be precise, and that applies to coding, talking, and even regular expressions. For example, you wouldn't write a grocery list for someone to Buy more .* because you would have no idea what you could get back. Instead you would write Buy more milk or Buy more bread, and in regular expressions, we can actually define these conditionals explicitly.

Specifically when using groups, you can use the | (logical OR, aka. the pipe) to denote different possible sets of characters. In the above example, I can write the pattern "Buy more (milk|bread|juice)" to match only the strings Buy more milk, Buy more bread, or Buy more juice.

Like normal groups, you can use any sequence of characters or metacharacters in a condition, for example, ([cb]ats*|[dh]ogs?) would match either cats or bats, or, dogs or hogs. Writing patterns with many conditions can be hard to read, so you should consider making them separate patterns if they get too complex.

https://regex101.com/

https://github.com/ziishaned/learn-regex#1-basic-matchers

Further reading:

https://www.regular-expressions.info/tutorial.html