Slawomir Chodnicki Follow

Nov 16, 2019 · 27 min read · Member-only · Listen

Everything you need to know about Regular Expressions

Photo by NESA by Makers on Unsplash

After reading this article you will have a solid understanding of what regular expressions are, what they can do, and what they can't do.

You'll be able to judge when to use them and — more importantly — when not to.

Let's start at the beginning.

What is a Regular Expression?

Say we have a list of all valid zip codes. Instead of keeping that long and unwieldy list around, it's often more practical to have a short and precise pattern that completely describes that set. Whenever you want to check whether a string is a valid zip code, you can match it against the pattern. You'll get a true or false result indicating whether the string belongs to the set of zip codes the regex pattern represents.

Let's expand on the set of zip codes. A list of zip codes is finite, consists of rather short strings, and is not particularly challenging computationally.

What about the set of strings that end in `.csv`? Can be quite useful when looking for data files. This set is infinite. You can't make a list up front. And the only way to test for membership is to go to the end of the string and compare the last four characters. Regular expressions are a way of encoding such patterns in a standardized way.

Regular expressions match sets of strings. Photo by Kristian Strand on Unsplash

The following is a regular expression pattern that represents our set of strings ending in `.csv`

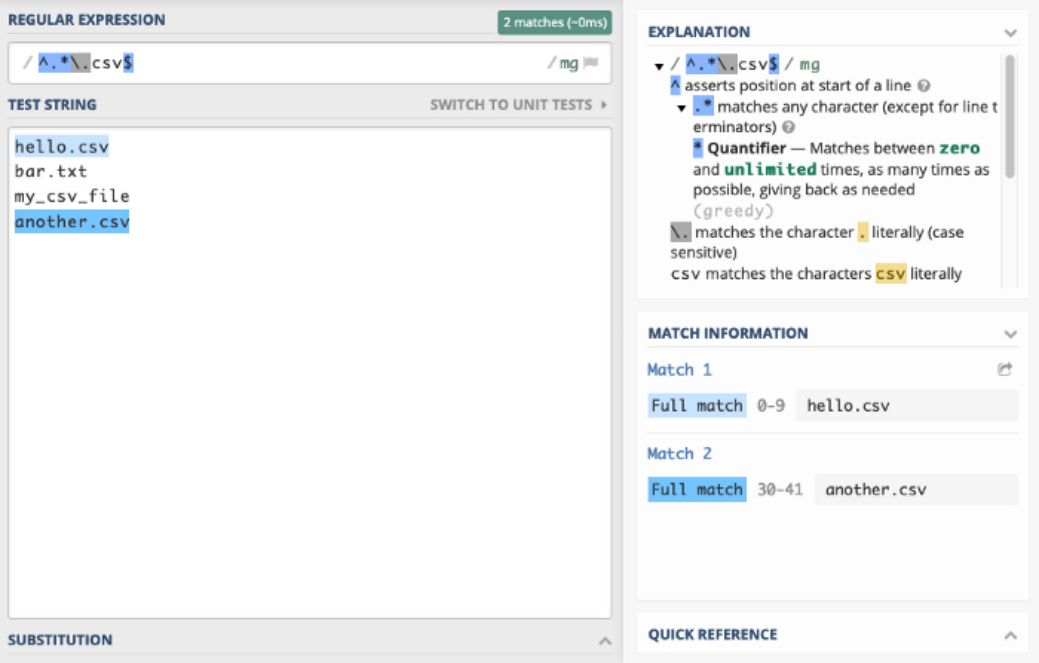
```
^.*\.csv$
```

matches. The above pattern matches `foo.csv`, but does not match `bar.txt` or `my_csv_file`.

Before you use regular expressions in your code, you can test them using an online regex evaluator, and experiment with a friendly UI.

I like regex101.com: you can pick the flavor of the regex engine, and patterns are nicely decomposed for you, so you get a good understanding of what your pattern actually does. Regex patterns can be cryptic.

I'd recommend you open regex101.com in another window or tab and experiment with the examples presented in this article interactively. You'll get a much better feel for regex patterns this way, I promise.



debugging regular expressions

What are Regular Expressions used for?

Regular expressions are useful in any scenario that benefits from full or partial pattern matches on strings. These are some common use cases:

- verify the structure of strings
- extract substrings form structured strings

All of these come up regularly when doing data preparation work.

The Building Blocks of a Regular Expression

A regular expression pattern is constructed from distinct building blocks. It may contain literals, character classes, boundary matchers, quantifiers, groups and the OR operator.

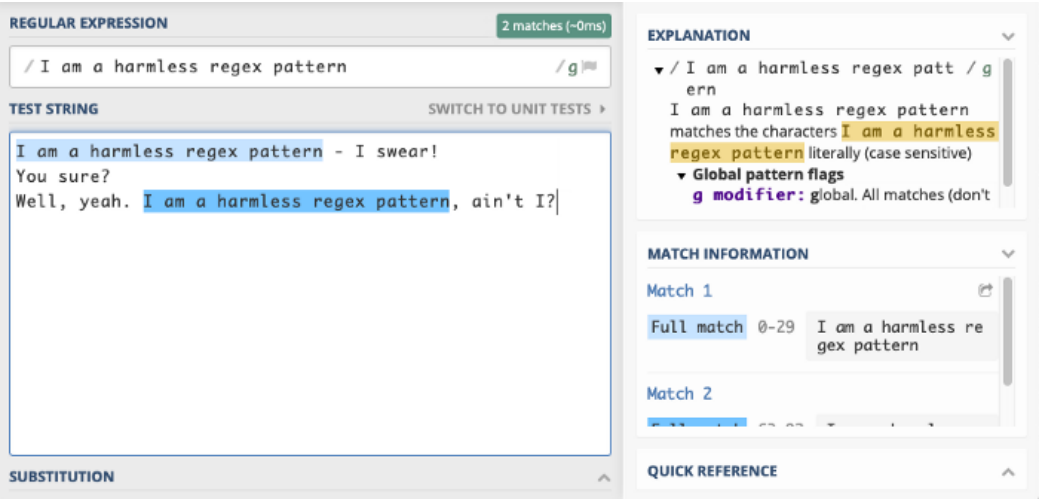
Let's dive in and look at some examples.

Literals

The most basic building block in a regular expression is a character a.k.a. literal. Most characters in a regex pattern do not have a special meaning, they simply match themselves. Consider the following pattern:

```
I am a harmless regex pattern
```

None of the characters in this pattern has special meaning. Thus each character of the pattern matches itself. Therefore there is only one string that matches this pattern, and it is identical to the pattern string itself.



matching a simple pattern

Escaping Literal Characters

What are the characters that do have special meaning? The following list shows characters that have special meaning in a regular expression. They must be

Character	Meaning	Escaped
^	boundary matcher	\^
$	boundary matcher	\$
\	escape character	\\
{	quantifier notation	\{
}	quantifier notation	\}
[	character class notation	\[
]	character class notation	\]
(	group notation	\(
)	group notation	\)
.	predefined character class	\.
*	quantifier	*
+	quantifier	\+
?	quantifier	\?
|	OR operator	|

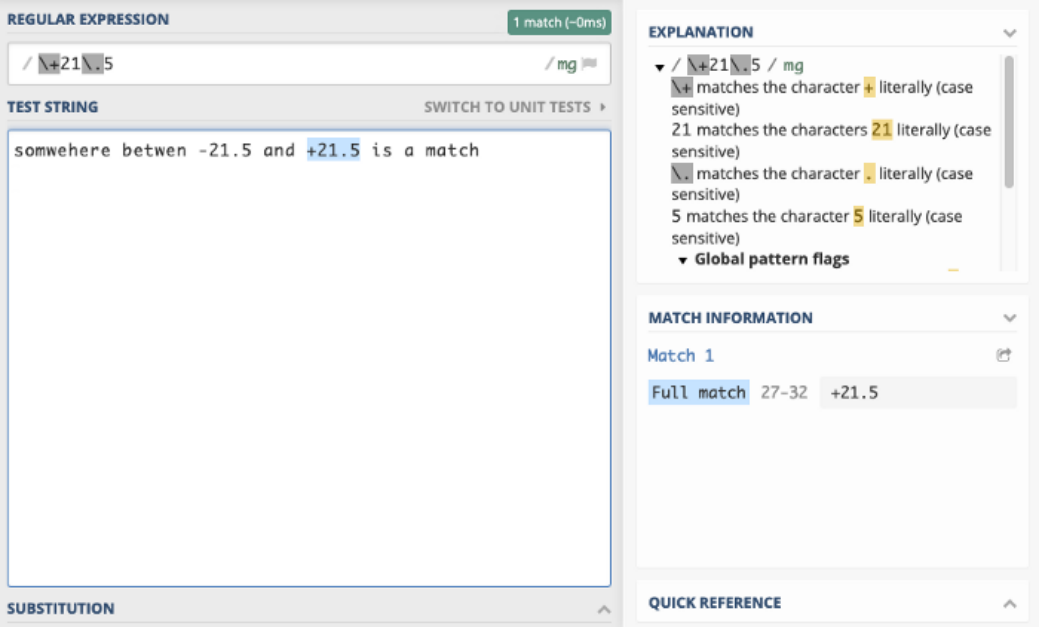
regex_non_literals.csv hosted with ❤ by GitHub view raw

characters with special meaning in regular expressions

Consider the following pattern:

```
\+21\.5
```

The pattern consists of literals only — the `+` has special meaning and has been escaped, so has the `.` — and thus the pattern matches only one string: `+21.5`



Matching non-printable Characters

It's best to use the proper escape sequences for them:

Search this file…

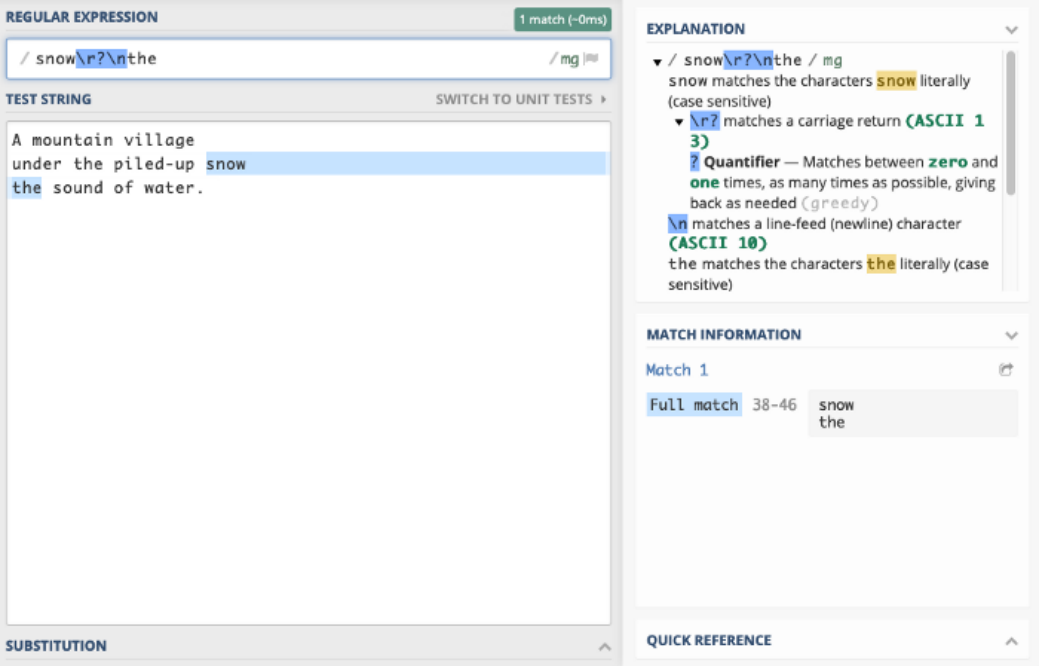
	Escape Sequence	Meaning
2	\t	The tab character ('\u0009')
3	\n	The newline (line feed) character ('\u000A')
4	\r	The carriage-return character ('\u000D')

regex_non_printable_characters.csv hosted with ❤ by GitHub view raw

If you need to match a line break, they usually come in one of two flavors:

- `\n` often referred to as the unix-style newline
- `\r\n` often referred to as the windows-style newline

To catch both possibilities you can match on `\r?\n` which means: optional `\r` followed by `\n`



matching a newline

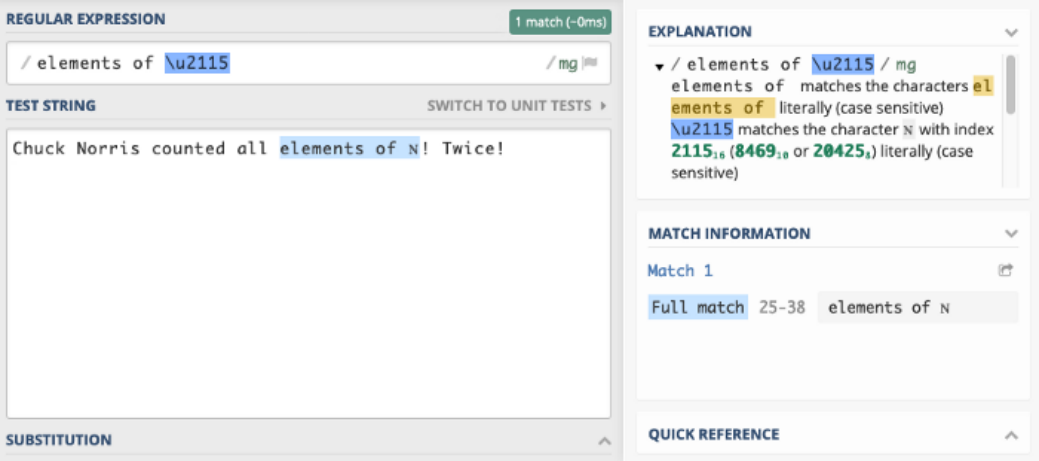
Matching any Unicode Character

Sometimes you have to match characters that are best expressed by using their Unicode index. Sometimes a character simply cannot be typed— like control

Sometimes your programming language simply does not support putting certain characters into patterns. Characters outside the BMP, such as 𝄞 or emojis are often not supported verbatim.

In many regex engines — such as Java, JavaScript, Python, and Ruby — you can use the `\uHexIndex` escape syntax to match any character by its Unicode index. Say we want to match the symbol for natural numbers: ℕ - U+2115

The pattern to match this character is: `\u2115`



matching a unicode symbol

Other engines often provide an equivalent escape syntax. In Go, you would use `\x{2115}` to match ℕ

Unicode support and escape syntax varies across engines. If you plan on matching technical symbols, musical symbols, or emojis — especially outside the BMP — check the documentation of the regex engine you use to be sure of adequate support for your use-case.

Escaping Parts of a Pattern

Sometimes a pattern requires consecutive characters to be escaped as literals. Say it's supposed to match the following string: `+???+`

The pattern would look like this:

```
\+\?\?\?\+
```

Depending on your regex engine, there might be a way to start and end a literal section in your pattern. Check your docs. In Java and Perl sequences of characters that should be interpreted literally can be enclosed by `\Q` and `\E`. The following pattern is equivalent to the above:

```
\Q+???+\E
```

Escaping parts of a pattern can also be useful if it is constructed from parts, some of which are to be interpreted literally, like user-supplied search words.

If your regex engine does not have this feature, the ecosystem often provides a function to escape all characters with special meaning from a pattern string, such as lodash escapeRegExp.

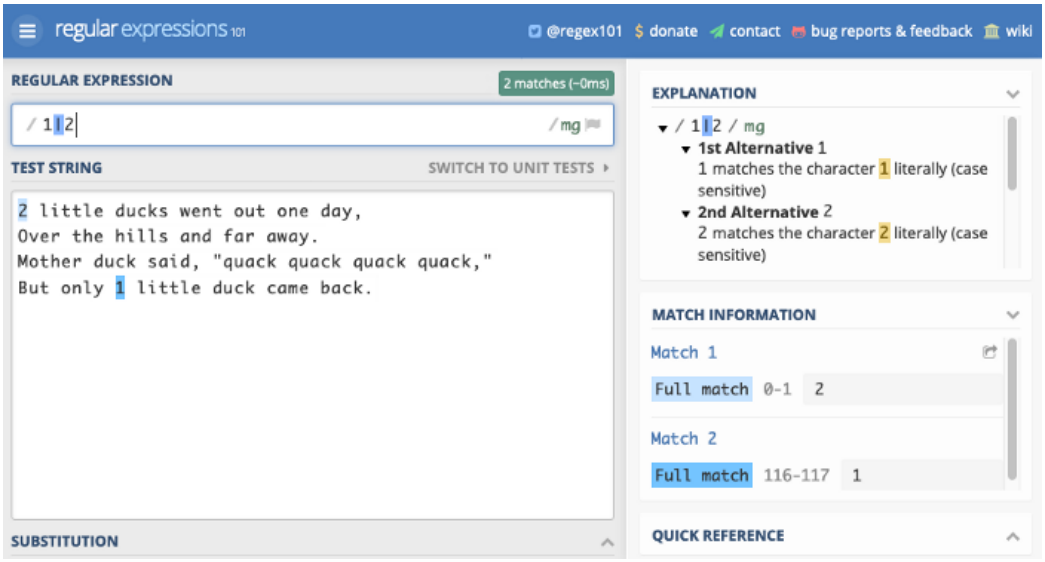
The OR Operator

The pipe character `|` is the selection operator. It matches alternatives. Suppose a pattern should match the strings `1` and `2`

The following pattern does the trick:

```
1|2
```

The patterns left and right of the operator are the allowed alternatives.



```
William Turner|Bill Turner
```

The second part of the alternatives is consistently `Turner`. Would be convenient to put the alternatives `William` and `Bill` up front, and mention `Turner` only once. The following pattern does that:

```
(William|Bill) Turner
```

It looks more readable. It also introduces a new concept: Groups.

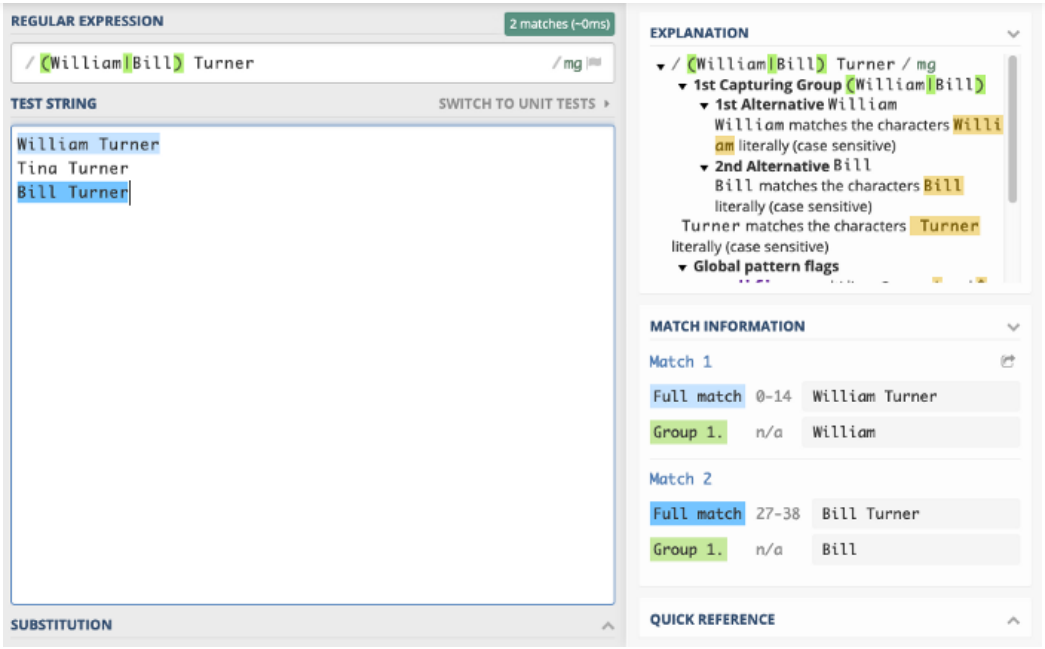
Groups

You can group sub-patterns in sections enclosed in round brackets. They group the contained expressions into a single unit. Grouping parts of a pattern has several uses:

- simplify regex notation, making intent clearer
- apply quantifiers to sub-expressions
- extract sub-strings matching a group
- replace sub-strings matching a group

Let's look at a regex with a group: `(William|Bill) Turner`

Groups are sometimes referred to as "capturing groups" because in case of a match, each group's matched sub-string is captured, and is available for extraction.



How captured groups are made available depends on the API you use. In JavaScript, calling `"my string".match(/pattern/)` returns an array of matches. The first item is the entire matched string and subsequent items are the sub-strings matching pattern groups in order of appearance in the pattern.

```
1  const pattern = /(Fox|Wolf) Force (Three|Four|Five)/;
2
3  const result = "Fox Force Five".match(pattern);
4
5  console.log(result)
6
7  // ["Fox Force Five", "Fox", "Five"]
```

regex_matching_simple_groups.js hosted with ❤ by GitHub view raw

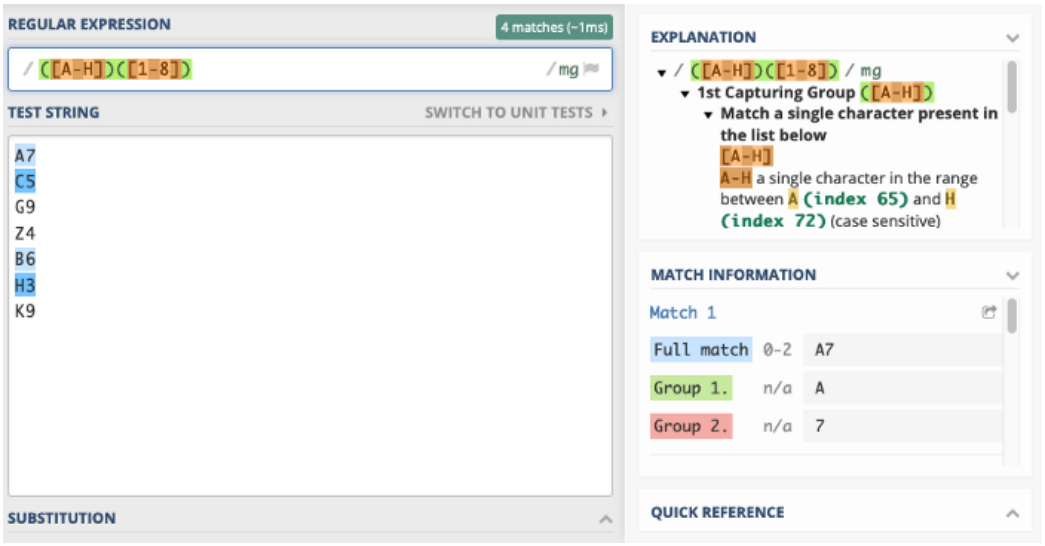
Accessing sub-strings in captured in groups

Example: Chess Notation

Consider a string identifying a chess board field. Fields on a chess board can be identified as A1-A8 for the first column, B1-B8 for the second column and so on until H1-H8 for the last column. Suppose a string containing this notation should be validated and the components (the letter and the digit) extracted using capture groups. The following regular expression would do that.

```
(A|B|C|D|E|F|G|H)(1|2|3|4|5|6|7|8)
```

```
([A-H])([1-8])
```



This sure looks more concise. But it introduces a new concept: Character Classes.

Character Classes

Character classes are used to define a set of allowed characters. The set of allowed characters is put in square brackets, and each allowed character is listed. The character class `[abcdef]` is equivalent to `(a|b|c|d|e|f)`. Since the class contains alternatives, it matches exactly one character.

The pattern `[ab][cd]` matches exactly 4 strings `ac`, `ad`, `bc`, and `bd`. It does *not* match `ab`, the first character matches, but the second character must be either `c` or `d`.

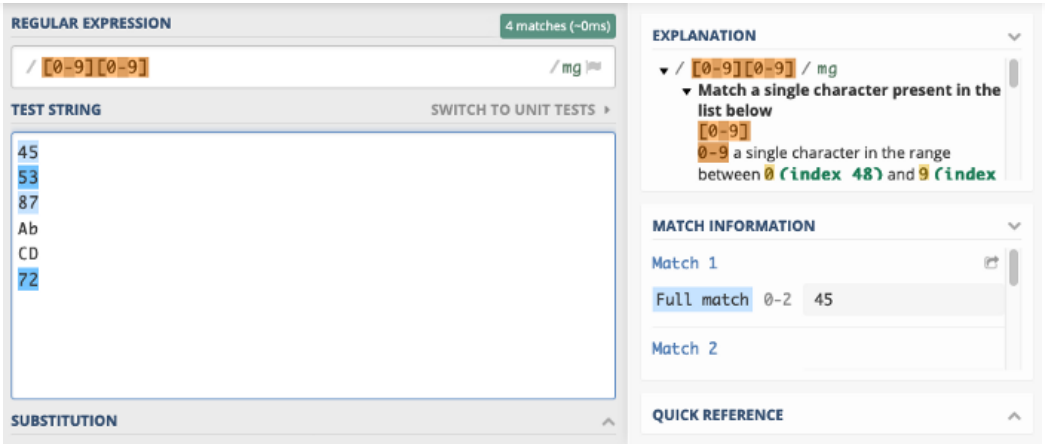
Suppose a pattern should match a two digit code. A pattern to match this could look like this:

```
[0123456789][0123456789]
```

This pattern matches all 100 two digit strings in the range from `00` to `99`.

Ranges

It is often tedious and error-prone to list all possible characters in a character



matching two characters in range 0–9

Characters are ordered by a numeric index— in 2019 that is almost always the Unicode index. If you're working with numbers, Latin characters and basic punctuation, you can instead look at the much smaller historical subset of Unicode: ASCII.

The digits zero through nine are encoded sequentially through code-points: `U+0030` for `0` to code point `U+0039` for `9`, so a character set of `[0–9]` is a valid range.

Lower case and upper case letters of the Latin alphabet are encoded consecutively as well, so character classes for alphabetic characters are often seen too. The following character set matches any lower case Latin character:

```
[a–z]
```

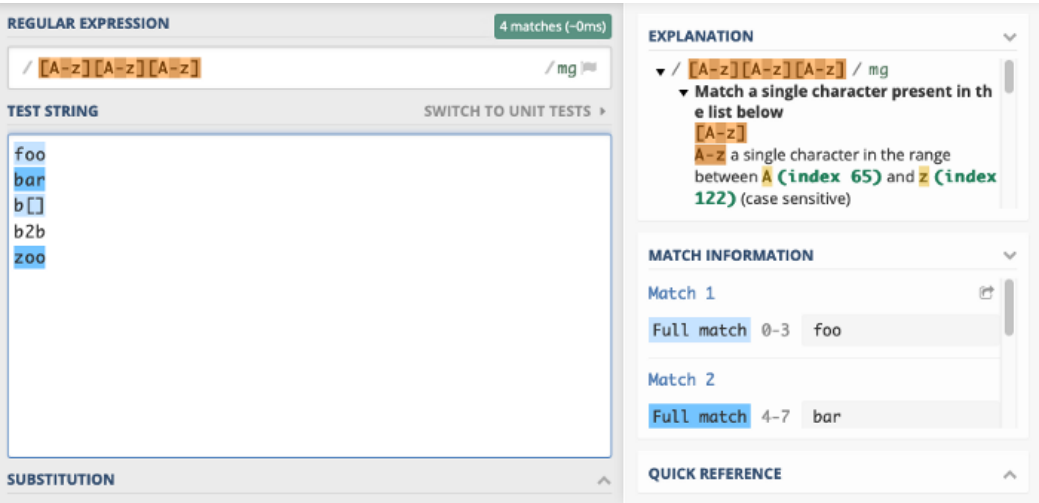
You can define multiple ranges within the same character class. The following character class matches all lower case and upper case Latin characters:

```
[A–Za–z]
```

You might get the impression that the above pattern could be abbreviated to:

```
[A–z]
```

That is a valid character class, but it matches not only A-Z and a-z, it also matches all characters defined between Z and a, such as `[`, `\`, and `^`.



the range A-z includes unexpected characters [and]

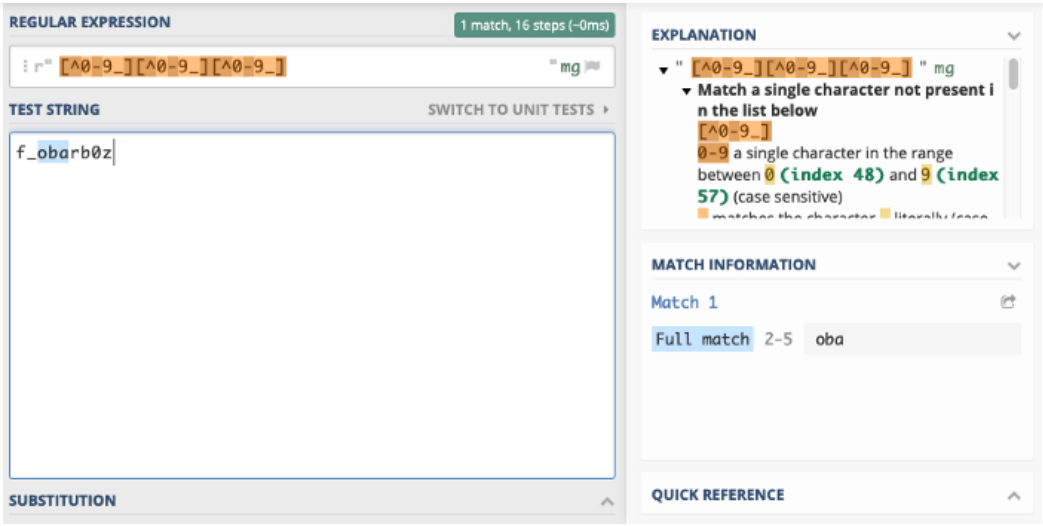
> *If you're tearing your hair out cursing the stupidity of the people who defined ASCII and introduce this mind-boggling discontinuity, hold your horses for a bit. ASCII was defined at a time where computing capacity was much more precious than today.*
>
> *Look at* `A hex: 0x41 bin: 0100 0001` *and* `a hex: 0x61 bin: 0110 0001`
>
> *How do you convert between upper and lower case? You flip* ***one*** *bit. That is true for the entire alphabet. ASCII is optimized to simplify case conversion. The people defining ASCII were very thoughtful. Some desirable qualities had to be sacrificed for others. You're welcome.*

You might wonder how to put the `-` character into a character class. After all, it is used to define ranges. Most engines interpret the `-` character literally if placed as the first or last character in the class: `[-+0-9]` or `[+0-9-]`. Some few engines require escaping with a backslash: `[\-+0-9]`

Negations

Sometimes it's useful to define a character class that matches most characters, except for a few defined exceptions. If a character class definition begins with a `^`, the set of listed characters is inverted. As an example, the following class allows any character as long as it's neither a digit nor an underscore.

```
[^0-9_]
```



looking for three consecutive non-digit and non-underscore characters

Please note that the `^` character is interpreted as a literal if it is not the first character of the class, as in `[f^o]`, and that it is a boundary matcher if used outside character classes.

Predefined Character Classes

Some character classes are used so frequently that there are shorthand notations defined for them. Consider the character class `[0–9]`. It matches any digit character and is used so often that there is a mnemonic notation for it: `\d`.

The following list shows character classes with most common shorthand notations, likely to be supported by any regex engine you use.

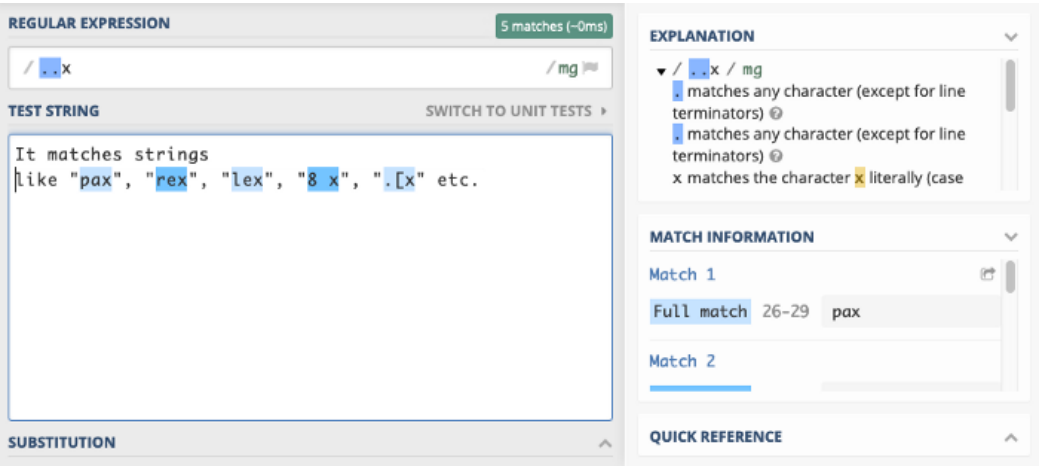
Most engines come with a comprehensive list of predefined character classes matching certain blocks or categories of the Unicode standard, punctuation, specific alphabets, etc. These additional character classes are often specific to the engine at hand, and not very portable.

The Dot Character Class

The most ubiquitous predefined character class is the dot, and it deserves a small section on its own. It matches any character except for line terminators like `\r` and `\n`.

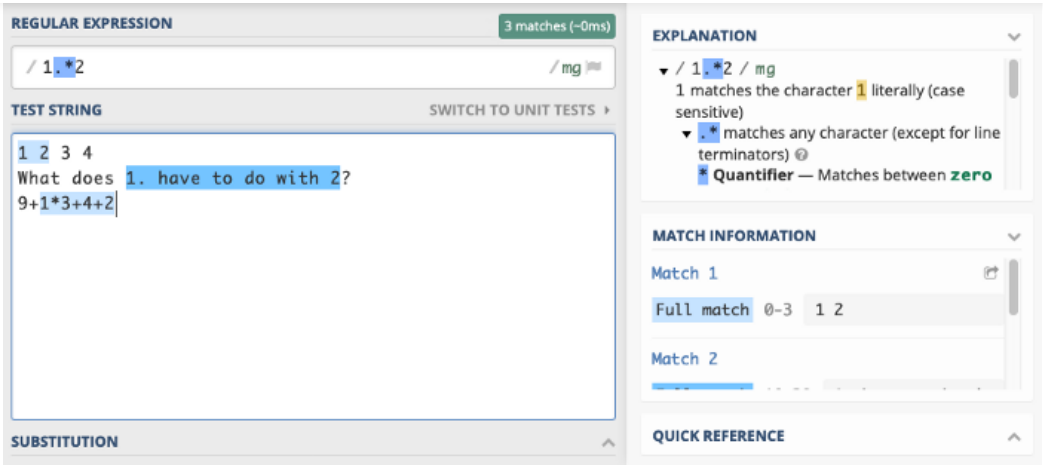
The following pattern matches any three character string ending with a lower case x:

```
..x
```



the dot matches anything except newline characters

In practice the dot is often used to create “anything might go in here” sections in a pattern. It is frequently combined with a quantifier and `.*` is used to match



matching anything between 1 and 2

Please note that the `.` character loses its special meaning, when used inside a character class. The character class `[.,]` simply matches two characters, the dot and the comma.

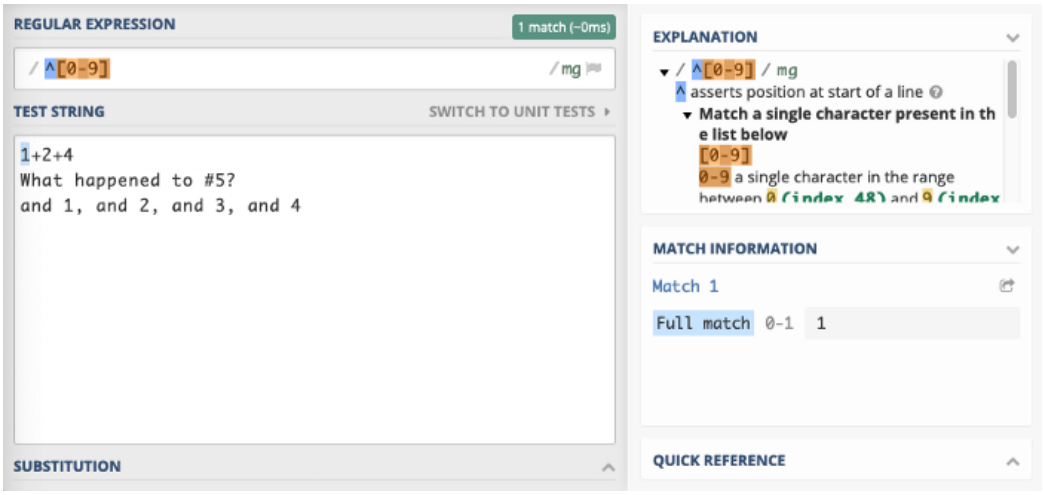
Depending on the regex engine you use you may be able to set the **dotAll** execution flag in which case `.` will match anything including line terminators.

Boundary Matchers

Boundary matchers — also known as “anchors” — do not match a character as such, they match a boundary. They match the positions between characters, if you will. The most common anchors are `^` and `$`. They match the beginning and end of a line respectively. The following table shows the most commonly supported anchors.

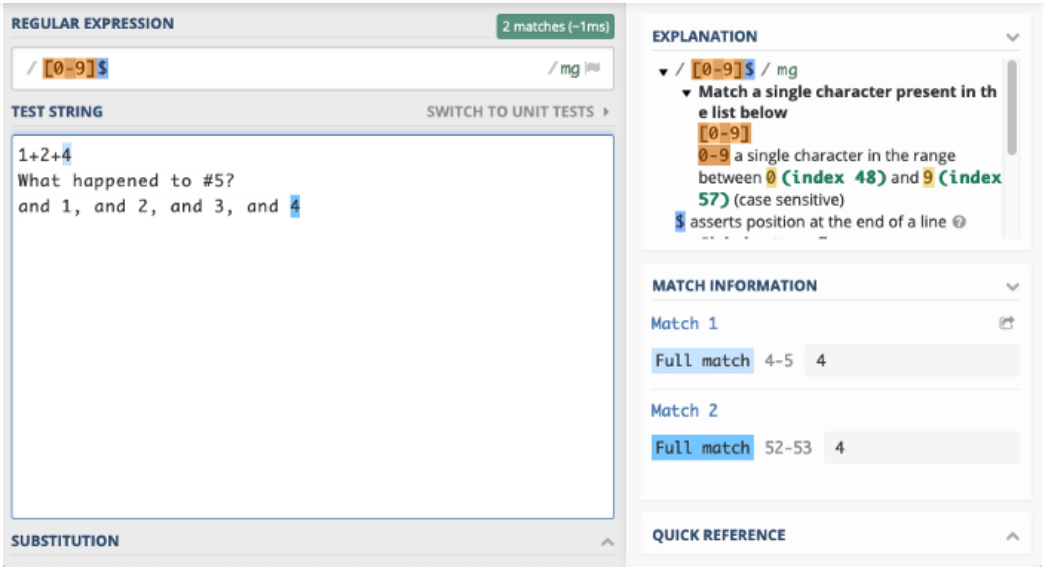
Anchoring to Beginnings and Endings

Consider a search operation for digits on a multi-line text. The pattern `[0–9]` finds every digit in the text, no matter where it is located. The pattern `^[0–9]` finds every digit that is the first character on a line.



matching digits at the beginning of a line

The same idea applies to line endings with `$`.



matching digits at the end of a line

The `\A` and `\Z` *or* `\z` anchors are useful matching multi-line strings. They anchor to the beginning and end of the entire input. The upper case `\Z` variant is tolerant of trailing newlines and matches just before that, effectively discarding any trailing newline in the match.

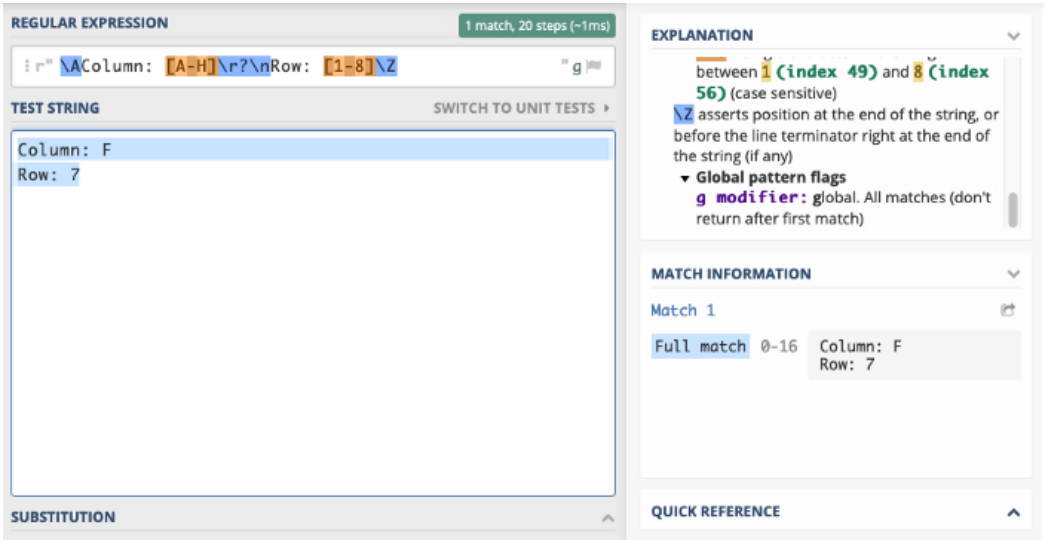
> *The \A and \Z anchors are supported by most mainstream regex engines, with the notable exception of JavaScript.*

Suppose the requirement is to check whether a text is a two-line record specifying a chess position. This is what the input strings looks like:

```
Column: F
Row: 7
```

The following pattern matches the above structure:

```
\AColumn: [A-H]\r?\nRow: [1-8]\Z
```

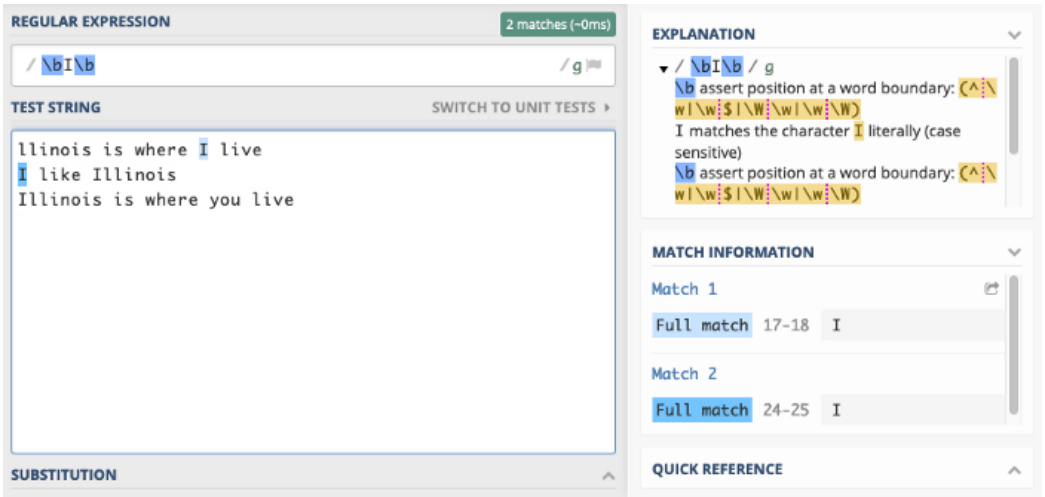


Using /A and /Z to anchor to beginning and end of input

Whole Word Matches

The `\b` anchor matches the edge of any alphanumeric sequence. This is useful if you want to do “whole word” matches. The following pattern looks for a standalone upper case `I`.

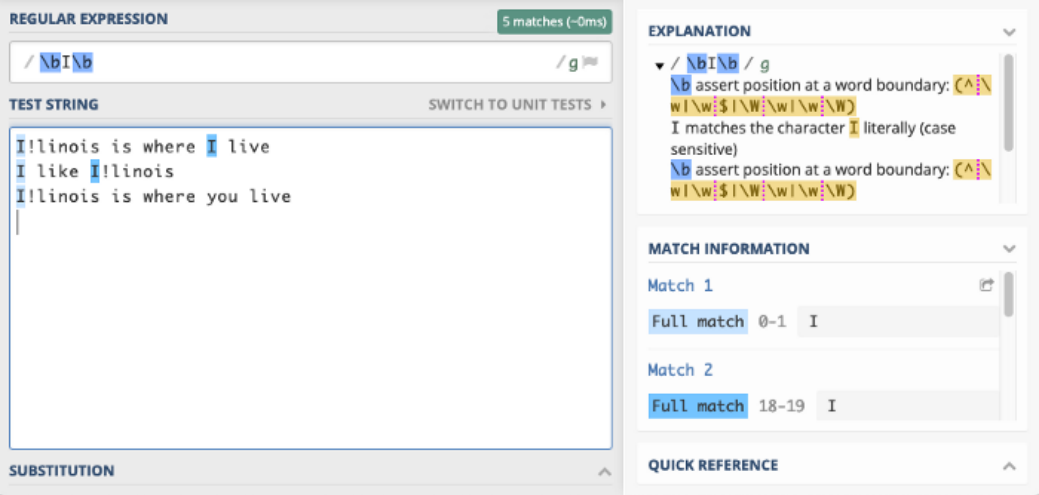
```
\bI\b
```



the \b anchor matches on transitions between “words”

The pattern does not match the first letter of `Illinois` because there is no word boundary to the right. The next letter is a word letter — defined by the character class `\w` as `[a-zA-Z0-9_]` —and not a non-word letter, which would constitute a boundary.

Let's replace `Illinois` with `I!linois`. The exclamation point is not a word character, and thus constitutes a boundary.



Misc Anchors

The somewhat esoteric non-word boundary `\B` is the negation of `\b`. It matches any position that is not matched by `\b`. It matches every position between characters *within* white space and alphanumeric sequences.

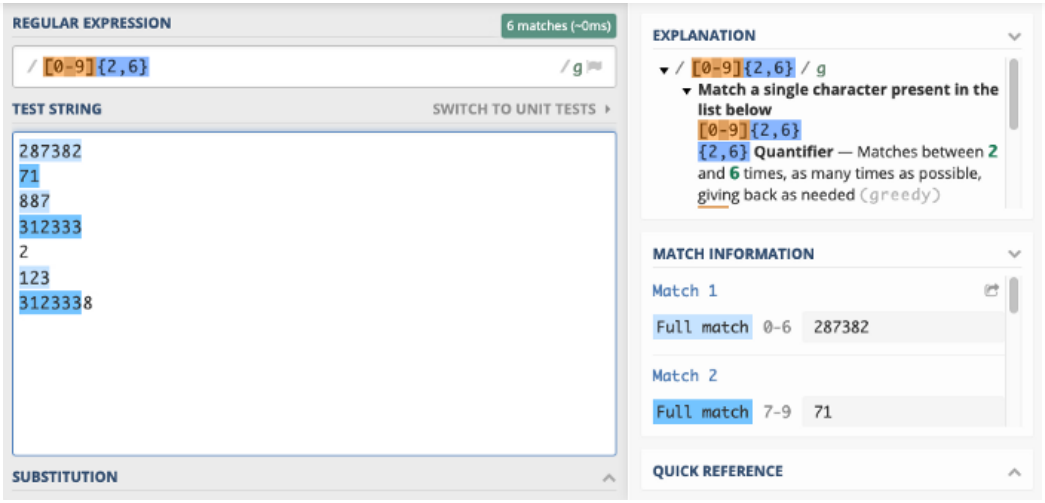
Some regex engines support the `\G` boundary matcher. It is useful when using regular expressions programmatically, and a pattern is applied repeatedly to a string, trying to find pattern all matches in a loop. It anchors to the position of the last match found.

Quantifiers

Any literal or character group matches the occurrence of exactly one character. The pattern `[0-9][0-9]` matches exactly two digits. Quantifiers help specifying the expected number of matches of a pattern. They are notated using curly braces. The following is equivalent to `[0-9][0-9]`

```
[0-9]{2}
```

The basic notation can be extended to provide upper and lower bounds. Say it's necessary to match between two and six digits. The exact number varies, but it must be between two and six. The following notation does that:



sequences of 2–6 digits are matched

The upper bound is optional, if omitted any number of occurrences equal to or greater than the lower bound is acceptable. The following sample matches two or more consecutive digits.

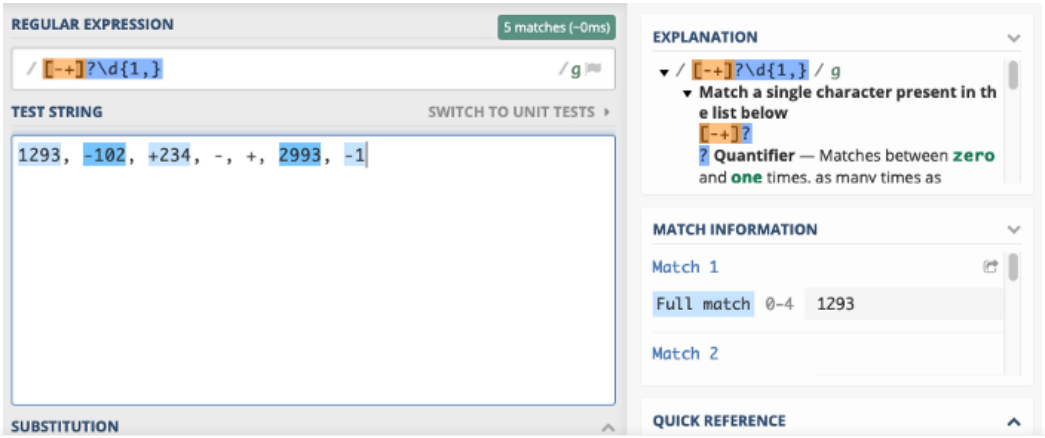
```
[0-9]{2,}
```

There are some predefined shorthands for common quantifiers that are very frequently used in practice.

The ? quantifier

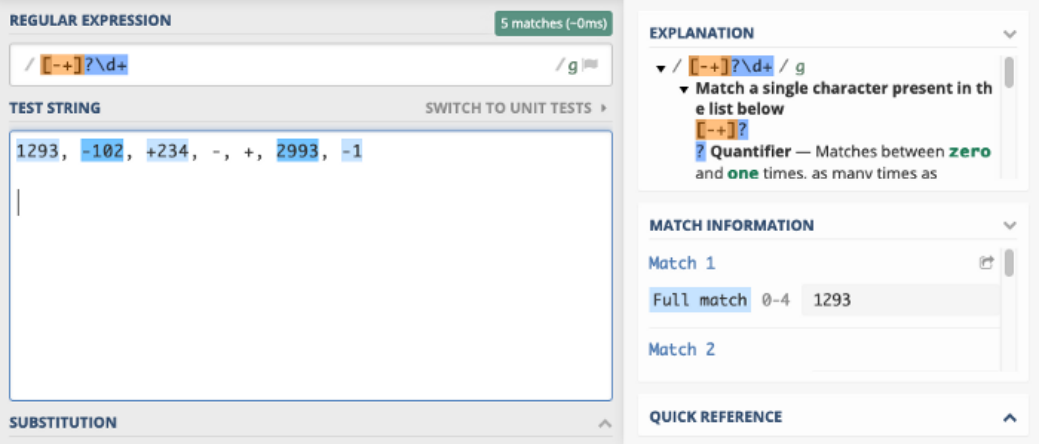
The `?` quantifier is equivalent to `{0, 1}`, which means: optional single occurrence. The preceding pattern may not match, or match once.

Let's find integers, optionally prefixed with a plus or minus sign: `[-+]?\d{1,}`



The `+` quantifier is equivalent to `{1,}`, which means: at least one occurrence.

We can modify our integer matching pattern from above to be more idiomatic by replacing `{1,}` with `+` and we get: `[-+]?\d+`



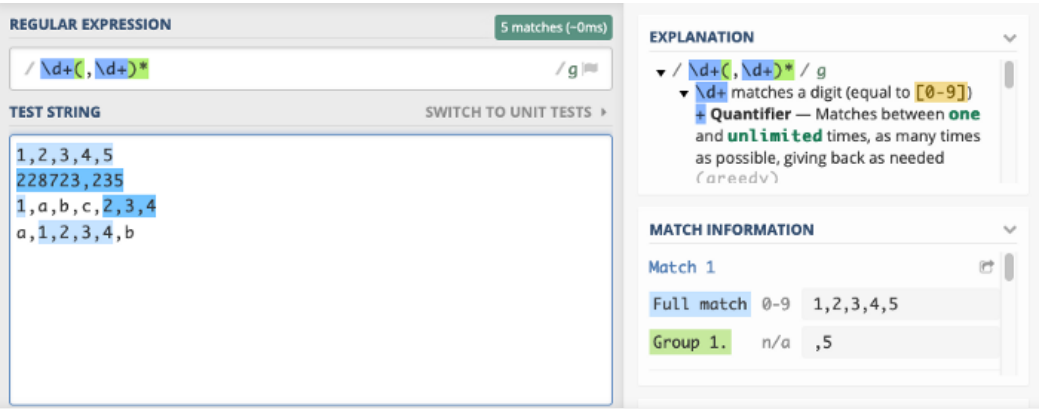
finding integers with optional sign again

The * quantifier

The `*` quantifier is equivalent to `{0,}`, which means: zero or more occurrences. You'll see it very often in conjunction with the dot as `.*`, which means: any character don't care how often.

Let's match an comma separated list of integers. Whitespace between entries is not allowed, and at least one integer must be present: `\d+(,\d+)*`

We're matching an integer followed by any number of groups containing a comma followed by an integer.

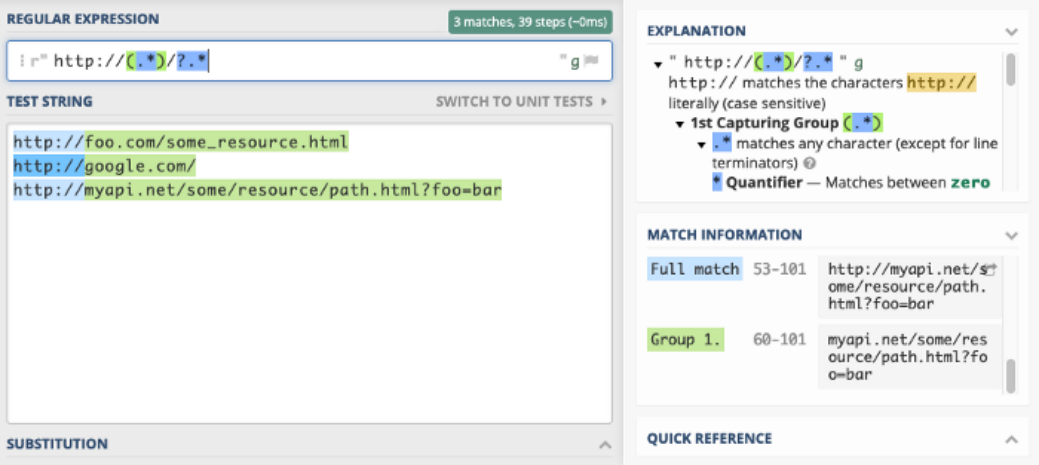


Suppose the requirement is to match the domain part from a http URL in a capture group. The following seems like a good idea: match the protocol, then capture the domain, then an optional path. The idea translates roughly to this:

```
http://(.*)/?
```

> *If you're using an engine that uses* `/regex/` *notation like JavaScript, you have to escape the forward slashes:* `http:\/\/(.*)\/?.*`

It matches the protocol, captures what comes after the protocol as domain and it allows for an optional slash and some arbitrary text after that, which would be the resource path.



greedy capture matches too much

Strangely enough, the following is captured by the group given some input strings:

unexpected portion of url captured by group

The results are somewhat surprising, as the pattern was designed to capture the domain part only, but it seems to be capturing everything till the end of the URL.

This happens because each quantifier encountered in the pattern tries to match as much of the string as possible. The quantifiers are called **greedy** for this reason.

Let's check the of matching behaviour of: `http://(.*)/?.*`

The greedy `*` in the capturing group is the first encountered quantifier. The `.` character class it applies to matches any character, so the quantifier extends to the end of the string. Thus the capture group captures everything. But wait, you say, there's the `/?.*` part at the end. Well, yes, and it matches what's left of the string — nothing, a.k.a the empty string — perfectly. The slash is optional, and is followed by zero or more characters. The empty string fits. The entire pattern matches just fine.

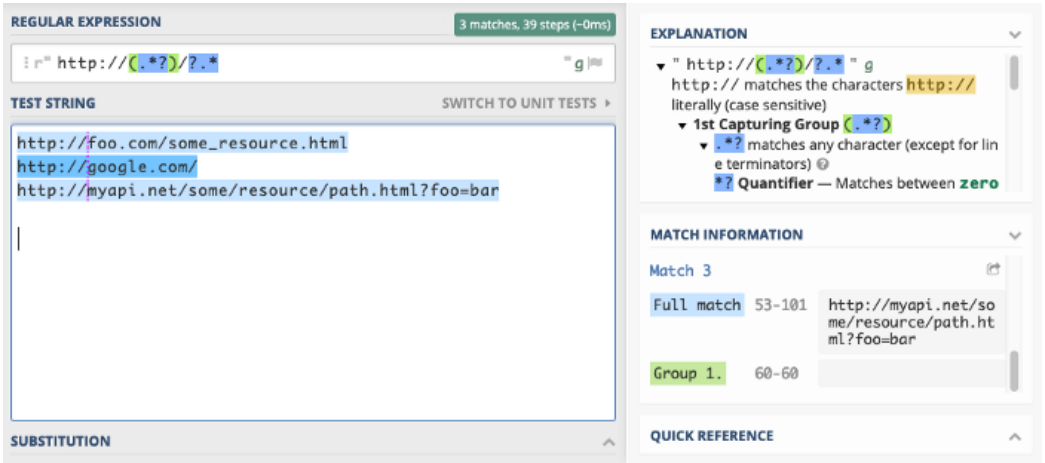
Alternatives to Greedy Matching

Greedy is the default, but not the only flavor of quantifiers. Each quantifier has a **reluctant** version, that matches the **least** possible amount of characters. The greedy versions of the quantifiers are converted to reluctant versions by appending a **?** to them.

The following table gives the notations for all quantifiers.

The quanfier `{n}` is equvalent in both greedy and reluctant versions. For the others the number of matched characters may vary. Let's revisit the example from above and change the capture group to match as little as possible, in the hopes of getting the domain name captured properly.

`http://(.*?)/?.*`

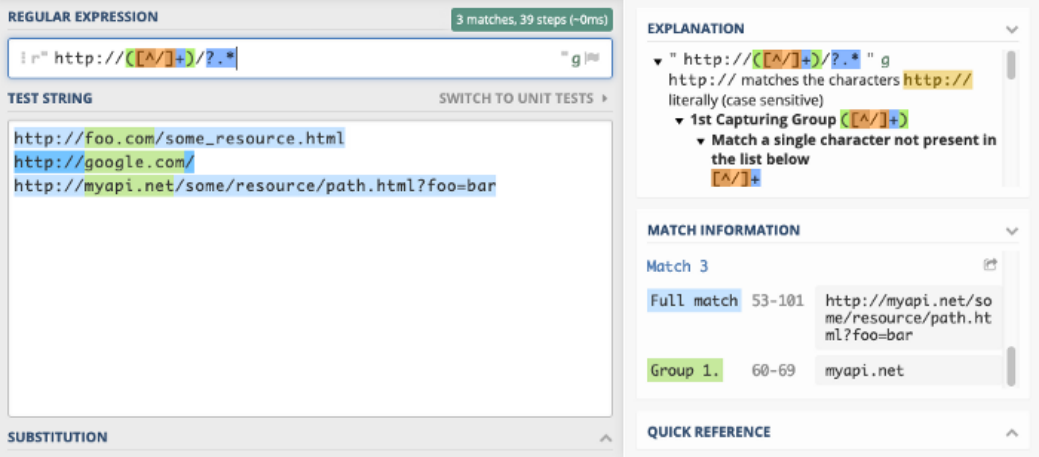


Using this pattern, nothing — more precisely the empty string — is captured by the group. Why is that? The capture group now captures as little as possible: nothing. The `(.*?)` captures nothing, the `/?` matches nothing, and the `.*` matches the entirety of what's left of the string. So again, this pattern does not work as intended.

So far the capture group matches too little or too much. Let's revert back to the greedy quantifier, but disallow the slash character in the domain name, and also require that the domain name be at least one character long.

```
http://([^/]+)/?.*
```

This pattern greedily captures one or more non slash characters after the protocol as the domain, and if finally any optional slash occurs it may be followed by any number of characters in the path.



Quantifier Performance

Both greedy and reluctant quantifiers imply some runtime overhead. If only a few such quantifiers are present, there are no issues. But if mutliple nested groups are each quantified as greedy or reluctant, determining the longest or shortest possible matches is a nontrivial operation that implies running back and forth on the input string adjusting the length of each quantifier's match to determine whether the expression as a whole matches.

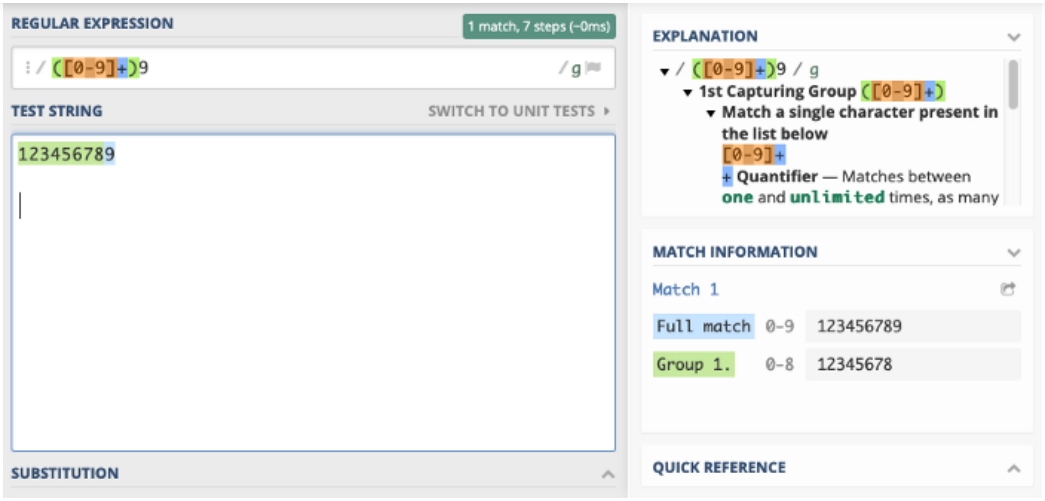
Pathological cases of catastrophic backtracking may occur. If performance or malicious input is a concern, it's best to prefer reluctant quantifiers and also have

Possessive quantifiers, if supported by your engine, act much like greedy quantifiers, with the distinction that they do not support backtracking. They try to match as many characters as possible, and once they do, they never yield any matched characters to accommodate possible matches for any other parts of the pattern.

They are notated by appending a `+` to the base greedy quantifier.

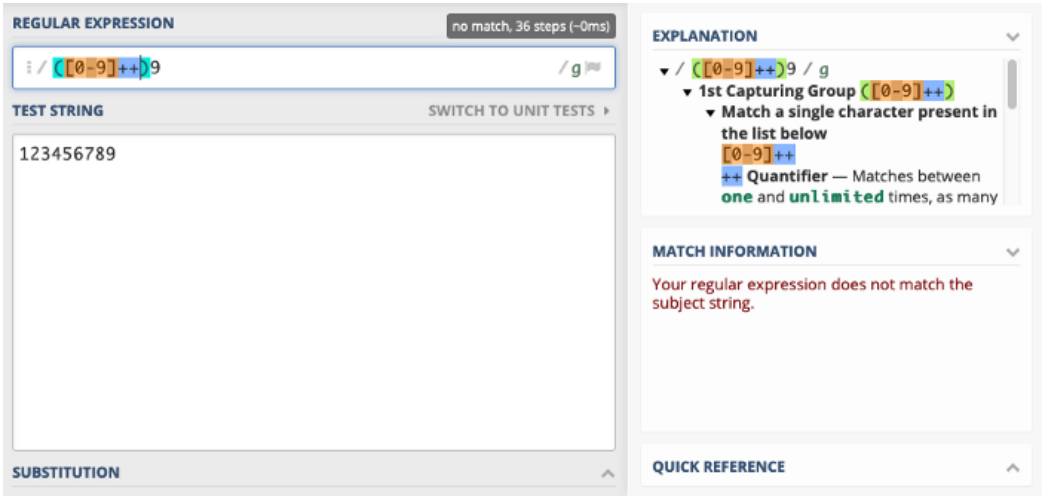
They are a fast performing version of "greedy-like" quantifiers, which makes them a good choice for performance sensitive operations.

Let's look at them in the PHP engine. First, let's look at simple greedy matches. Let's match some digits, followed by a nine: `([0–9]+)9`

greedy match

Now, when we replace the greedy with the possessive quantifier, it will match the entire input, then refuse to give back the `9` to avoid backtracking, and that will cause the entire pattern to not match at all.



possessive quantifer causing a non-match

When would you want possessive behaviour? When you know that you always want the longest conceivable match.

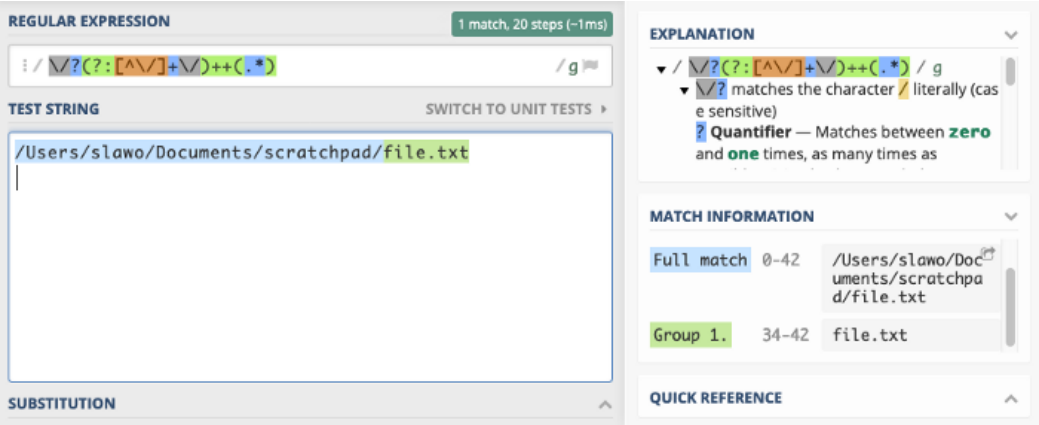
Let's say you want to extract the filename part of filesystem paths. Let's assume `/` as the path separator. Then what we effectively want is the last bit of the string after the last occurrence of a `/`.

A possessive pattern works well here, because we always want to consume all

```
\/?(?:[^\/]+\/)++(.*)
```

> *Note: using PHP `/regex/` notation here, so the forward slashes are escaped.*

We want to allow absolute paths, so we allow the input to start with an optional forward slash. We then possessively consume folder names consisting of a series of non-slash characters followed by a slash. I've used a non-capturing group for that — so it's notated as `(?:pattern)` instead of just `(pattern)`. Anything that is left over after the last slash is what we capture into a group for extraction.



example of posessive matching

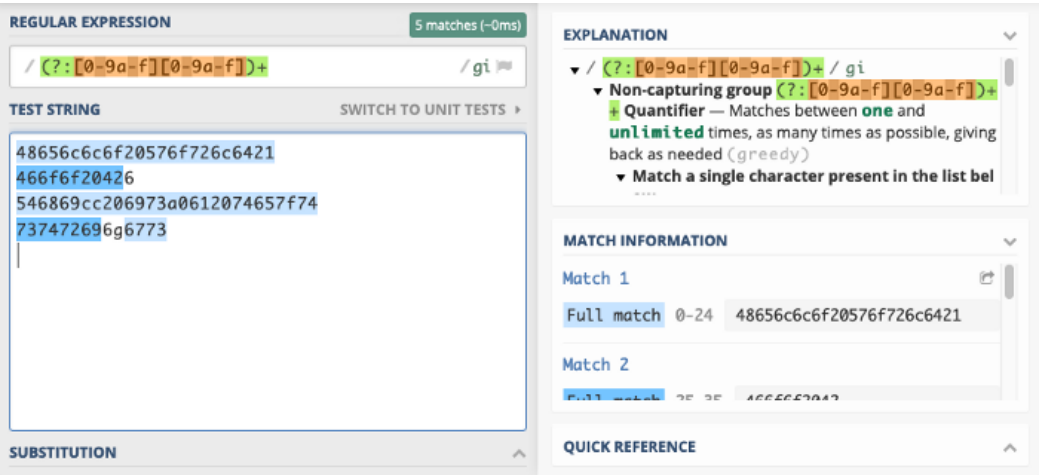
Non-Capturing Groups

Non-capturing groups match exactly the way normal groups do. However, they do not make their matched content available. If there's no need to capture the content, they can be used to improve matching performance. Non-capturing groups are written as: `(?:pattern)`

Suppose we want to verify that a hex-string is valid. It needs to consist of an even number of hexadecimal digits each between `0–9` or `a–f`. The following expression does the job using a group:

```
([0-9a-f][0-9a-f])+
```

Since the point of the group in the pattern is to make sure that the digits come in pairs, and the digits actually matched are not of any relevance, the group may just as well be replaced with the faster performing non-capturing group:



matching sequences of hex bytes

Atomic Groups

There is also a fast-performing version of a non-capturing group, that does not support backtracking. It is called the “independent non-capturing group” or “atomic group”.

It is written as `(?>pattern)`

An atomic group is a non-capturing group that can be used to optimize pattern matching for speed. Typically it is supported by regex engines that also support possessive quantifiers.

Its behavior is also similar to possessive quantifiers: once an atomic group has matched a part of the string, that first match is permanent. The group will never try to re-match in another way to accommodate other parts of the pattern.

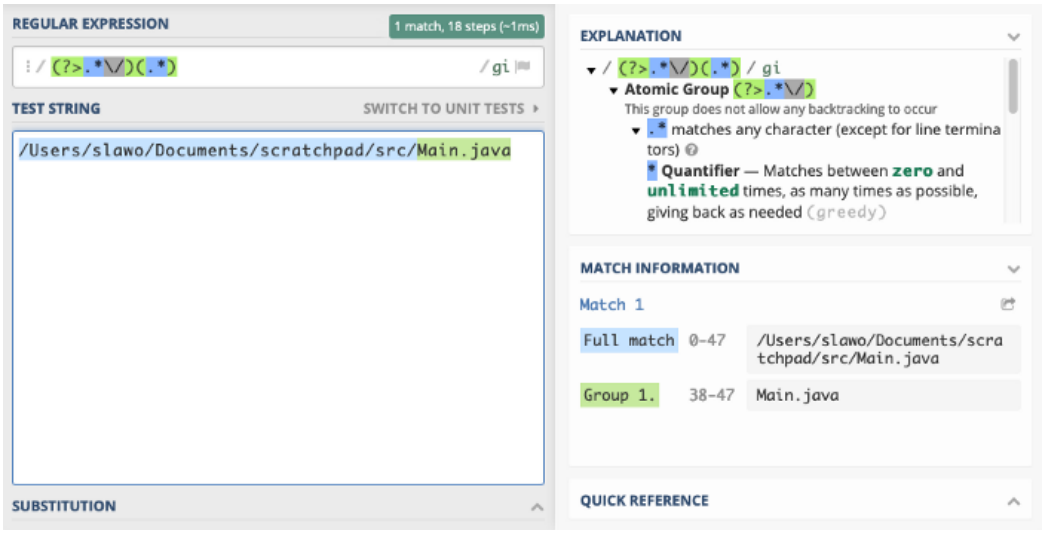
`a(?>bc|b)c` matches `abcc` but it does not match `abc`.

The atomic group’s first successful match is on `bc` and it stays that way. A normal group would re-match during backtracking to accommodate the `c` at the end of the pattern for a successful match. But an atomic group’s first match is permanent, it won’t change.

This is useful if you want to match as fast as possible, and don’t want any backtracking to take place anyway.

```
(?>.*\/)(.*)
```

> *Note: using PHP* `/regex/` *notation here, so the forward slashes are escaped.*



atomic group matching

A normal group would have done the job just as well, but eliminating the possibility of backtracking improves performance. If you're matching millions of inputs against non-trivial regex patterns, you'll start noticing the difference.

It also improves resilience against malicious input designed to DoS-Attack a service by triggering catastrophic backtracking scenarios.

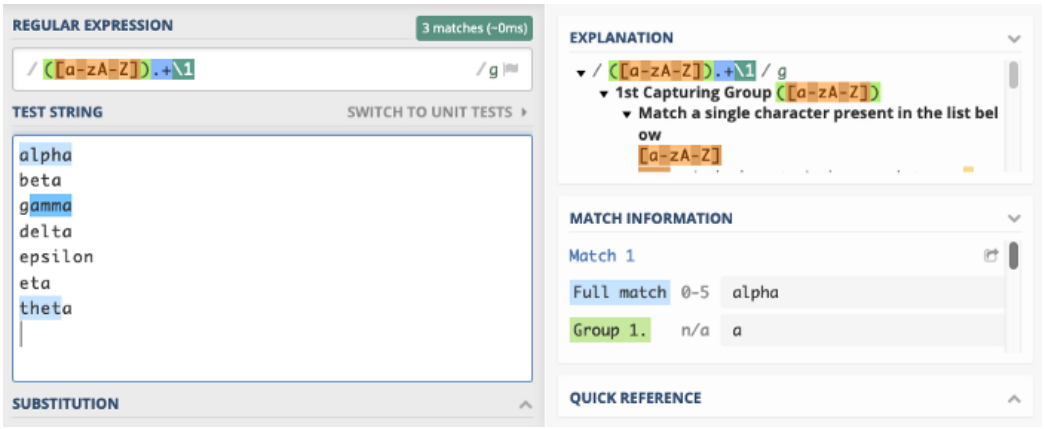
Back References

Sometimes it's useful to refer to something that matched earlier in the string. Suppose a string value is only valid if it starts and ends with the same letter. The words "alpha", "radar", "kick", "level" and "stars" are examples. It is possible to capture part of a string in a group and refer to that group later in the pattern pattern: a back reference.

Back references in a regex pattern are notated using `\n` syntax, where `n` is the number of the capture group. The numbering is left to right starting with 1. If groups are nested, they are numbered in the order their opening parenthesis is encountered. Group 0 always means the entire expression.

```
([a-zA-Z]).+\1
```

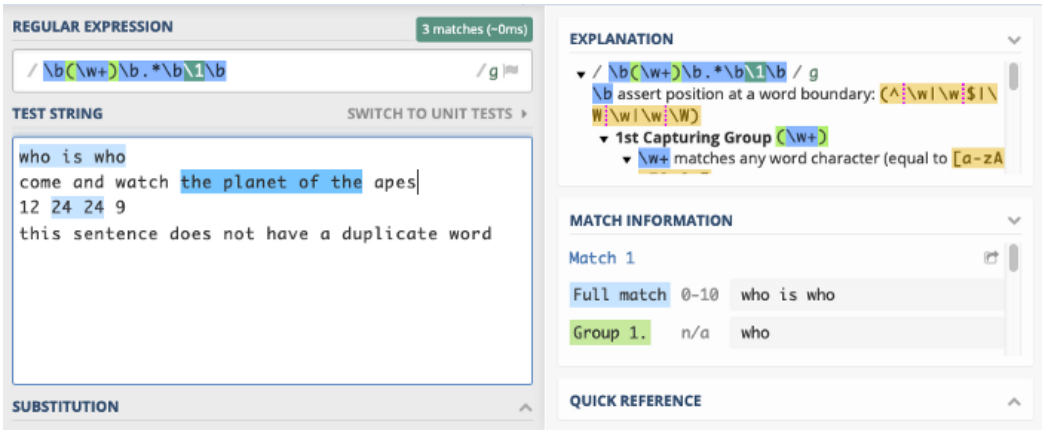
In words: an lower or upper case letter — that letter is captured into a group — followed by any non-empty string, followed by the letter we captured at the beginning of the match.



on-letter back references

Let's expand a bit. An input string is matched if it contains any alphanumeric sequence — think: word — more then once. Word boundaries are used to ensure that whole words are matched.

```
\b(\w+)\b.*\b\1\b
```



Search and Replace with Back References

Regular expressions are useful in search and replace operations. The typical use case is to look for a sub-string that matches a pattern and replace it with

These back references effectively allow to rearrange parts of the input string.

Consider the following scenario: the input string contains an A-Z character prefix followed by an optional space followed by a 3–6 digit number. Strings like `A321`, `B86562`, `F 8753`, and `L 287`.

The task is to convert it to another string consisting of the number, followed by a dash, followed by the character prefix.

Input	Output
A321	321–A
B86562	86562–B
F 8753	8753–F
L 287	287–L

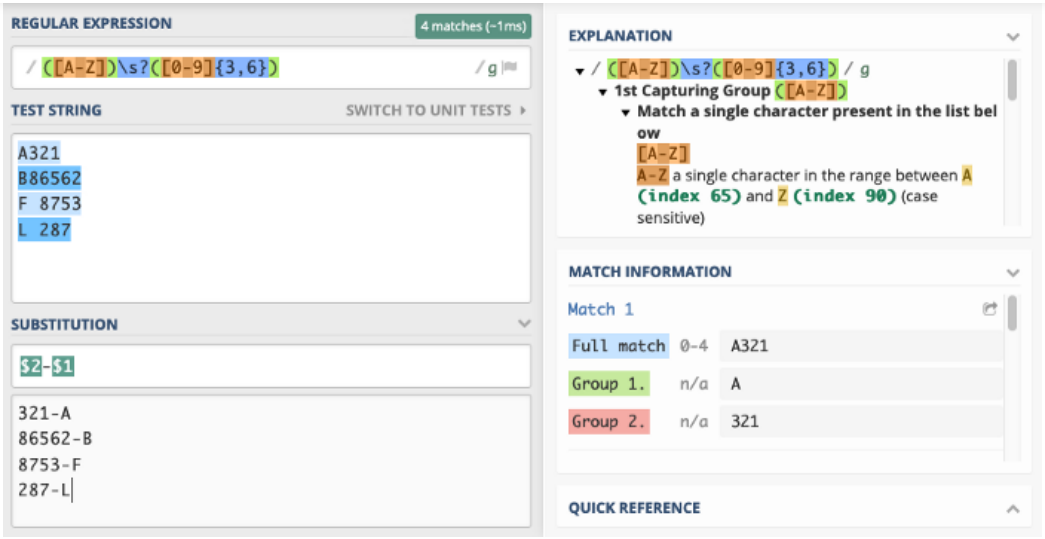
The first step to transform one string to the other is to capture each part of the string in a capture group. The search pattern looks like this:

```
([A-Z])\s?([0-9]{3,6})
```

It captures the prefix into a group, allows for an optional space character, then captures the digits into a second group. Back references in a replacement string are notated using `$n` syntax, where `n` is the number of the capture group. The replacement string for this operation should first reference the group containing the numbers, then a literal dash, then the first group containing the letter prefix. This gives the following replacement string:

```
$2–$1
```

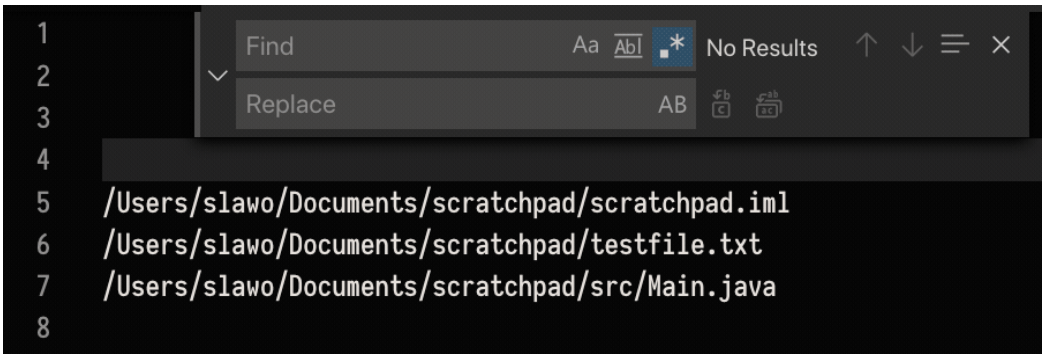
Thus `A321` is matched by the search pattern, putting `A` into `$1` and `312` into `$2`. The replacement string is arranged to yield the desired result: The number comes first, then a dash, then the letter prefix.



Please note that, since the `$` character carries special meaning in a replacement string, it must be escaped as `$$` if it should be inserted as a character.

This kind of regex-enabled search and replace is often offered by text editors. Suppose you have a list of paths in your editor, and the task at hand is to prefix the file name of each file with an underscore. The path `/foo/bar/file.txt` should become `/foo/bar/_file.txt`

With all we learned so far, we can do it like this:

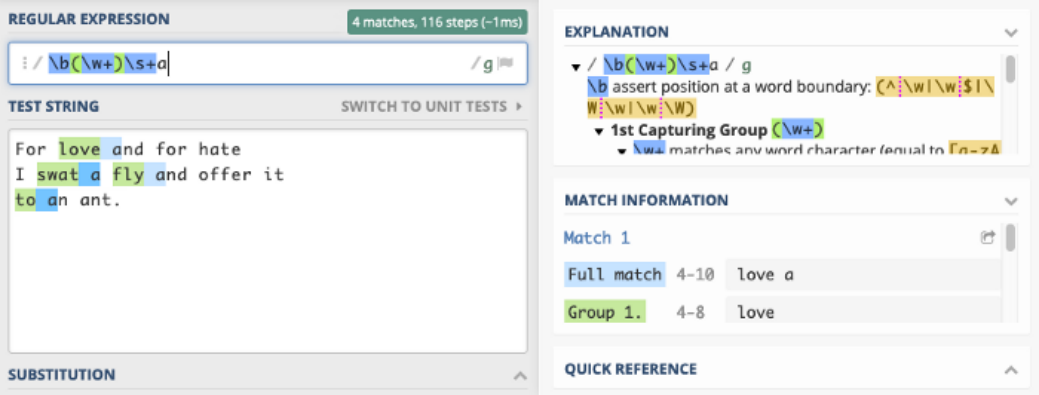


example regex-enabled search and replace in VS Code

Look, but don't touch: Lookahead and Lookbehind

It is sometimes useful to assert that a string has a certain structure, without actually matching it. How is that useful?

Let's try `\b(\w+)\s+a` it anchors to a word boundary, and matches word characters until it sees some space followed by an a.



first attempt at matching words that are followed by words beginning with an a

In the above example, we match `love`, `swat`, `fly`, and `to`, but fail to capture the `an` before `ant`. This is because the `a` starting `an` has been consumed as part of the match of `to`. We've scanned past that `a`, and the word `an` has no chance of matching.

Would be great if there was a way to assert properties of the first character of the next word without actually consuming it.

Constructs asserting existence, but not consuming the input are called "lookahead" and "lookbehind".

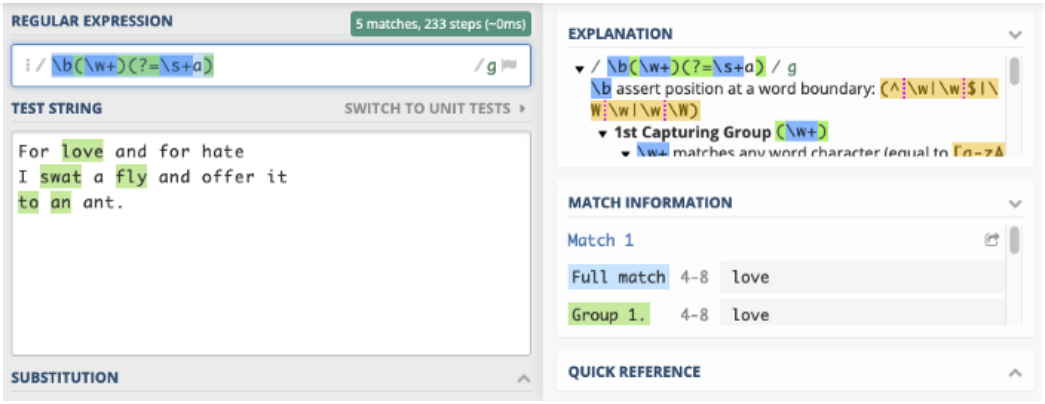
Lookahead

Lookaheads are used to assert that a pattern matches ahead. They are written as `(?=pattern)`

Let's use it to fix our pattern:

```
\b(\w+)(?=\s+a)
```

We've put the space and initial `a` of the next word into a lookahead, so when scanning a string for matches, they are checked but not consumed.



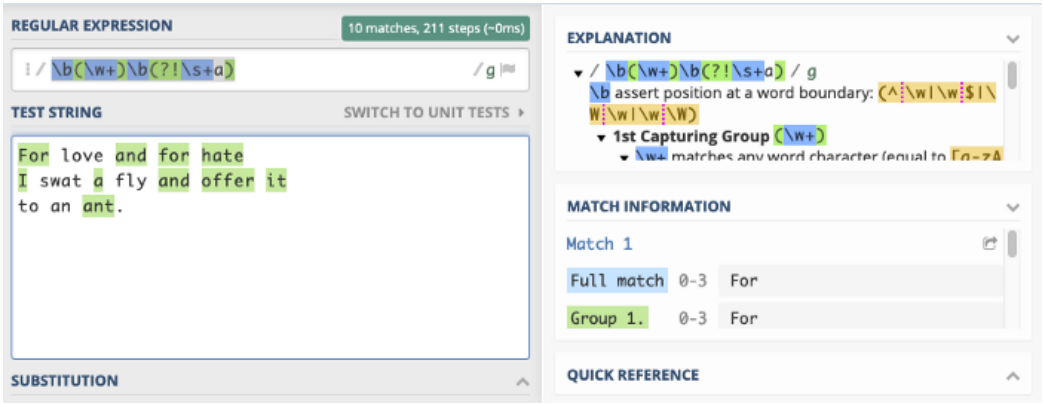
look ahead asserts a pattern matches ahead, but does not consume it

A negative lookahead asserts that its pattern does not match ahead. It is notated as `(?!pattern)`

Let's find all words not followed by a word that starts with an `a`.

```
\b(\w+)\b(?!\s+a)
```

We match whole words which are not followed by some space and an `a`.



negative look ahead asserts that its pattern does not match ahead

Lookbehind

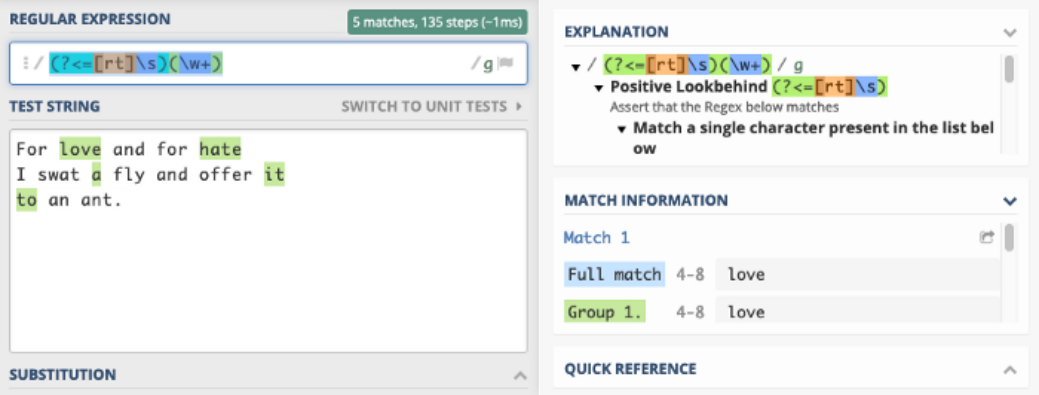
The lookbehind serves the same purpose as the lookahead, but it applies to the left of the current position, not to the right. Many regex engines limit the kind of pattern you can use in a lookbehind, because applying a pattern backwards is something that they are not optimized for. Check your docs!

A lookbehind is written as `(?<=pattern)`

It asserts the existence of something before the current position. Let's find all words that come after a word ending with an `r` or `t`.

```
(?<=[rt]\s)(\w+)
```

We assert that there is an `r` or `t` followed by a space, then we capture the sequence of word characters that follows.



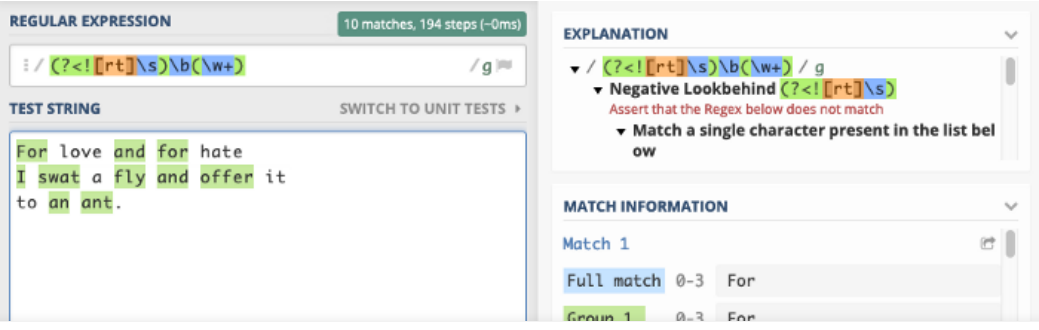
a lookbehind to capture certain words

There's also a negative lookbehind asserting the non-existence of a pattern to the left. It is written as `(?<!pattern)`

Let's invert the words found: We want to match all words that come after words not ending with `r` or `t`.

```
(?<![rt]\s)\b(\w+)
```

We match all words by `\b(\w+)`, and by prepending `(?<![rt]\s)` we ensure that any words we match are not preceded by a word ending in `r` or `t`.



Split patterns

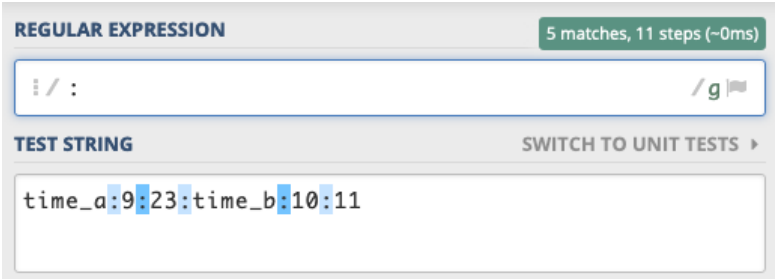
If you're working with an API that allows you to split a string by pattern, it is often useful to keep lookaheads and lookbehinds in mind.

A regex split typically uses the pattern as a delimiter, and removes the delimiter from the parts. Putting lookahead or lookbehind sections in a delimiter makes it match without removing the parts that were merely looked at.

Suppose you have a string delimited by `:`, in which some of the parts are labels consisting of alphabetic characters, and some are time stamps in the format `HH:mm`.

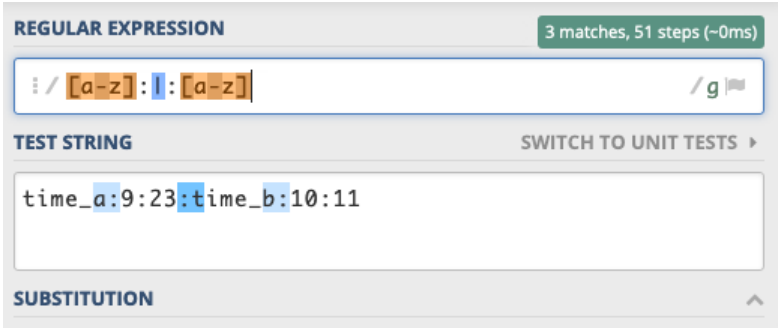
Let's look at input string `time_a:9:23:time_b:10:11`

If we just split on `:`, we get the parts: `[time_a, 9, 32, time_b, 10, 11]`



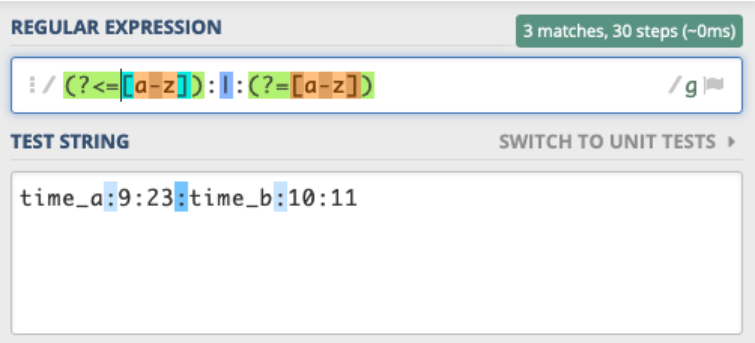
splitting on delimiter

Let's say we want to improve by splitting only if the `:` has a letter on either side. The delimiter is now `[a-z]:|:[a-z]`



including adjacent letters in the match treats them as part of the delimiter

If we refine the delimiter to use lookahead and lookbehind for the adjacent characters, their existence will be verified, but they won't match as part of the delimiter: `(?<=[a-z]):|:(?=[a-z])`



putting the characters into lookbehind and lookahead does not consume them

Finally we get the parts we want: `[time_a, 9:32, time_b, 10:11]`

Regex Modifiers

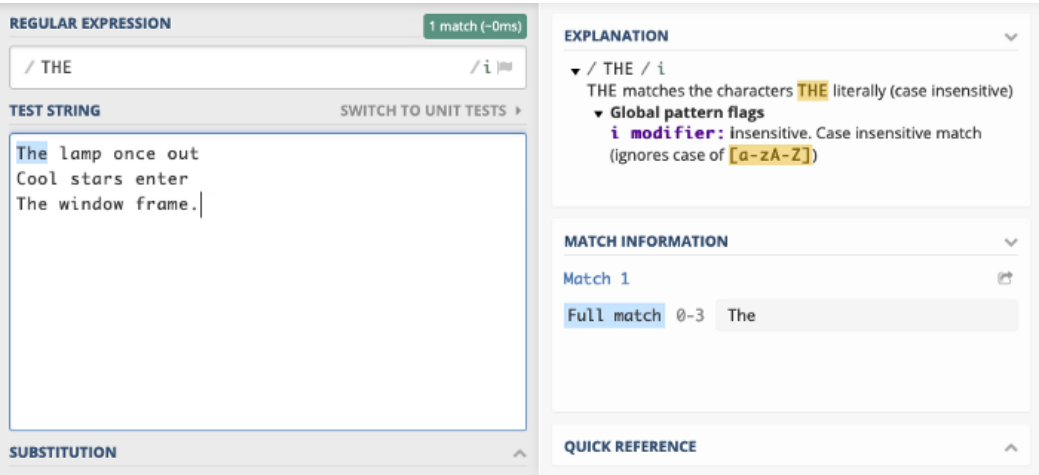
Most regex engines allow setting flags or modifiers to tune aspects of the pattern matching process. Be sure to familiarise yourself with the way your engine of choice handles such modifiers.

They often make the difference between a impractically complex pattern and a trival one.

You can expect to find case (in-)sensitivity modifiers, anchoring options, full match vs. patial match mode, and a dotAll mode which lets the `.` character class match anything including line terminators.

JavaScript, Python, Java, Ruby, .NET

Let's look at JavaScript, for example. If you want case insensitive mode and only the first match found, you can use the `i` modifier, and make sure to omit the `g` modifier.



case insensitive match, with only the first match found

Limitations of Regular Expressions

Arriving at the end of this article you may feel that all possible string parsing problems can be dealt with, once you get regular expressions under your belt.

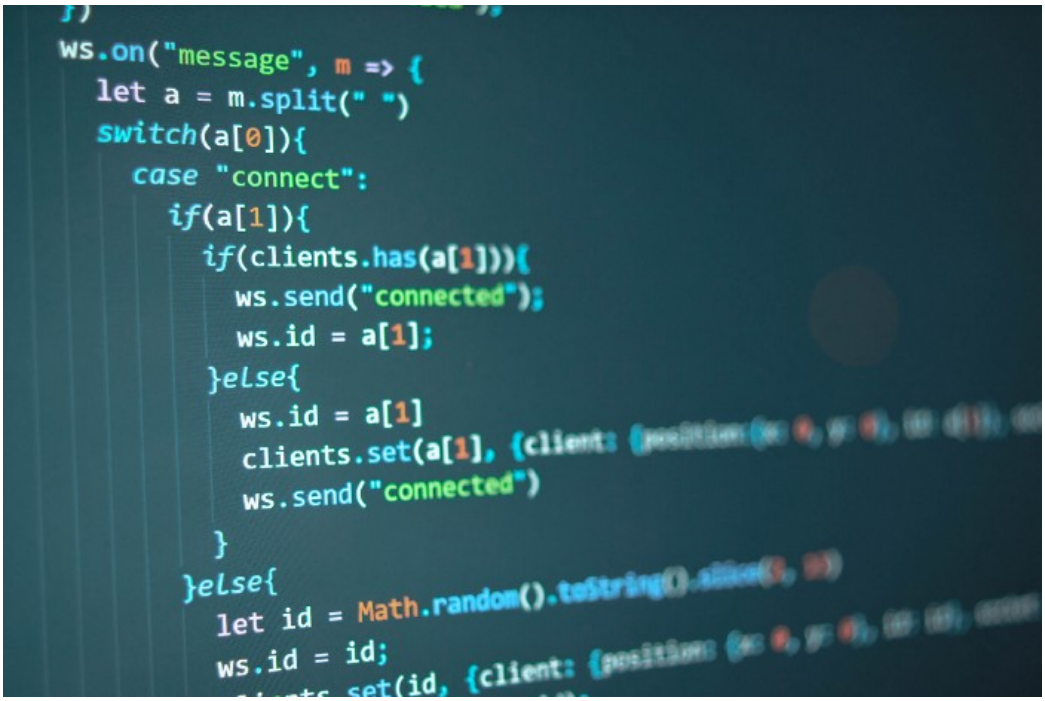
Well, no.

This article introduces regular expressions as a shorthand notation for sets of strings. If you happen to have the exact regular expression for zip codes, you have a shorthand notation for the set of all strings representing valid zip codes. You can easily test an input string to check if it is an element of that set. There is a problem however.

> There are many meaningful sets of strings for which there is no regular expression!

The set of valid JavaScript programs has no regex representation, for example. There will never be a regex pattern that can check if a JavaScript source is syntactically correct.

This is mostly due to regex' inherent inability to deal with nested structures of arbitrary depth. Regular expressions are inherently non-recursive. XML and JSON are nested structures, so is the source code of many programming languages. Palindromes are another example— words that read the same forwards and backwards like rgcaagr — are a very simple form of nested



If yor input can arbitrarily nest like JavaScript, you can't validate it with Regular Expressions alone. Photo by Christopher Robin Ebbinghaus on Unsplash

You can construct patterns that will match nested structures up to a certain depth but you can't write a pattern that matches arbitrary depth nesting.

Nested structures often turn out to be not regular. If you're interested in computation theory and classifications of languages — that is, sets of strings — have a glimpse at the Chomsky Hiararchy, Formal Grammars and Formal Languages.

Know when to reach for a different Hammer

Let me conclude with a word of caution. I sometimes see attempts trying to use regular expressions not only for lexical analysis — the identification and extraction of tokens from a string — but also for semantic analysis trying to interpret and validate each token's meaning as well.

While lexical analysis is a perfectly valid use case for regular expressions, attempting semantic validation more often than not leads towards creating another problem.

The plural of "regex" is "regrets"

Suppose a string shall be an IPv4 address in decimal notation with dots separating the numbers. A regular expression should validate that an input string is indeed an IPv4 address. The first attempt may look something like this:

```
([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})
```

It matches four groups of one to three digits separated by a dot. Some readers may feel that this pattern falls short. It matches `111.222.333.444` for example, which is not a valid IP address.

If you now feel the urge to change the pattern so it tests for each group of digits that the encoded number be between 0 and 255 — with possible leading zeros — then you're on your way to creating the second problem, and regrets.

Trying to do that leads away from lexical analysis — identifying four groups of digits — to a semantic analysis verifying that the groups of digits translate to admissible numbers.

This yields a dramatically more complex regular expression, examples of which is found here. I'd recommend solving a problem like this by capturing each group of digits using a regex pattern, then converting captured items to integers and validating their range in a separate logical step.

Use regular expressions — like all your tools — wisely. Photo by Todd Quackenbush on Unsplash

When working with regular expressions, the trade-off between complexity, maintainability, performance, and correctness should always be a conscious decision. After all, a regex pattern is as “write-only” as computing syntax can get. It is difficult to read regular expression patterns correctly, let alone debug and extend them.

My advice is to embrace them as a powerful string processing tool, but to neither overestimate their possibilities, nor the ability of human beings to handle them.

When in doubt, consider reaching for another hammer in the box.