Regular Expressions | A Complete Beginners Tutorial

Atmanand Nagpure

Jun 12, 2019 · 24 min read

Credit: unsplash.com

In the early days of computing, text processing and text pattern matching was a great deal as well as a huge challenge. There were no standards or pattern matching engines designed at that time that was easy-to-use and efficient. Matching text patterns or replacing a specific character pattern in a bulk sized file was an extremely difficult task at that time.

Until in the 1950s, an American mathematician named **Stephen Kleene** invented **Regular Expressions** which entirely revolutionalized text processing, pattern

What are Regular Expressions?

Regular Expressions (also called Regex or Regexp) is a pattern in which the rules for matching text are written in form of metacharacters, quantifiers or plain text. They are strings in which “what to match” is defined or written. Regex is used for finding patterns or replacing the matched patterns. It is used in almost all professional text editors, Integrated Development Environments(IDE s) and text processing applications.

What are the uses of Regular Expressions?

As their task is to find and/or replace the given pattern, they can be used in a lot of places where pattern matching is the prior task. Some of its major applications are listed as follows:

- Text Editors
- Search Engines and Search Mechanism back-ends of websites and APIs
- Code Editors and IDE s(linting, syntax highlighting)
- Data Entry Software
- Form and User Input data validation
- Data Analytics, Web Scraping

…and many more.

Regex Engines:

Regex engines are APIs written to perform regular expression operations. There are different kinds of regular expression engines, which have different kinds of features. Most of the regex syntax is the same, but there are some differences. Some engines have more features and some have less. Some of the popular regex engines are listed below:

- GNU ERE
- JavaScript Regex Engine

- Perl
- PHP preg
- XRegexp2
- C# Regex Engine
- Java Regex Engine

What does this tutorial offer?

This regex tutorial will give you a basic idea of what regular expressions are and how you can implement and use them in your regular tasks. As they are a great pattern matching tool, they'll also help you speed up your workflow. You will learn how to write your own regular expressions to match a pattern of your choice.

Let's get started!

The basic topics which will be covered in this tutorial:

1. Basic String Matching
2. OR Operation — (|) and [ ]
3. Ranging and Bracket Expressions ([a-z], etc…)
4. Quantifiers — * + {} ?
5. Character Classes — \d, \w, \s, and .
6. Anchors — ^ and $
7. Grouping and Capturing
8. Word Boundaries
9. Backreferencing
10. Greedy and Lazy

The Basics

order. Let's see it with an example.

Example:

```
hello*
```

In this string, the character **'o'** is preceding the character **'*'**. Because it comes before the character **'*'**. Another character 'e' is preceding the character 'l'.

Q. What is the meaning of the word following?

Answer: The word ***following*** means anything that is coming ***after*** something in order. Let's see it with an example.

Example:

```
MaxLinda
```

In this string, the character **'L'** is following the character **'x'**. Because the letter 'L' comes after the letter **'x'**. Another character **'d'** is following the character 'n'.

. . .

* Escaping:

Regular expression contains reserved metacharacters like [], (), ., etc. They have their special meaning in regular expressions. But, what if our input string contains any of those characters? How do we match those? Because if we try directly to match them by directly entering them, they will change the meaning of the regular expression and won't match what we require to match.

Example:

```
This is a [bracket].
```

To Match:

```
[bracket].
```

Solution:

So, we use the concept of escaping the characters. If we want to match any regular expression metacharacter in our input string, we can use a BACKSLASH ' \ ' character to escape that metacharacter. It will directly be interpreted as a regular character or symbol. Ex: If you want to match a square bracket '[', you can write a regular expression like '\['. It will be interpreted as a regular square bracket character.

Regular Expression:

```
\[\w+\]\.
```

> *Note: DO NOT TRY TO UNDERSTAND THIS REGEX RIGHT NOW. WE WILL LEARN ABOUT IT IN FURTHER SECTIONS.*

Tools Used:

Text Editor: **Atom text editor**

RegexBuddy: Another tool used in this tutorial is **RegexBuddy**. It is a great tool for writing and testing regular expressions. It is like an **IDE for regular expressions**.

. . .

1. Basic String Matching

So, this is an absolute basic way to match a pattern; by **manually writing out** the **string** that you want to match.

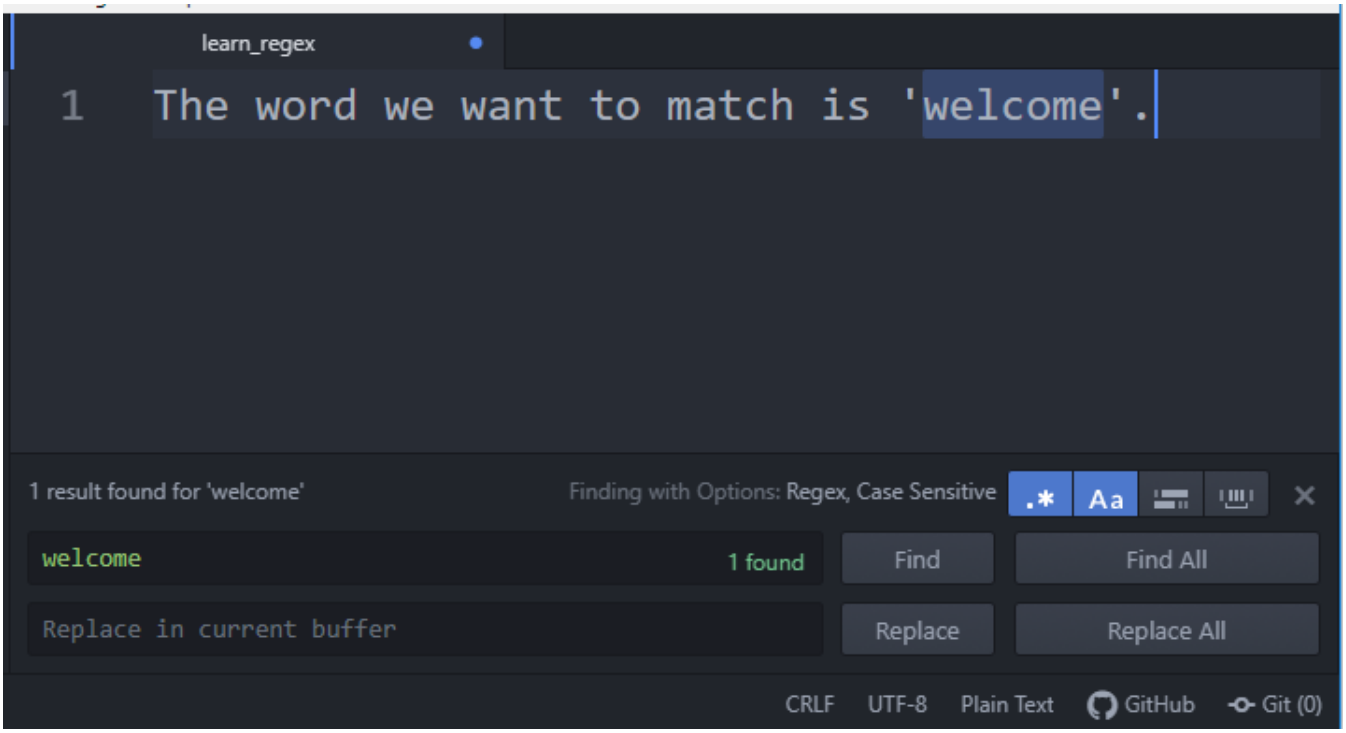
Input String:

Regular Expression:

```
welcome
```

Matched Pattern:

```
welcome
```



Basic String Matching

. . .

2. OR Operator — | and [ ]

Now, the OR operator means an obligation. It is used to match either of the strings supplied to it. The given example demonstrates it's usage.

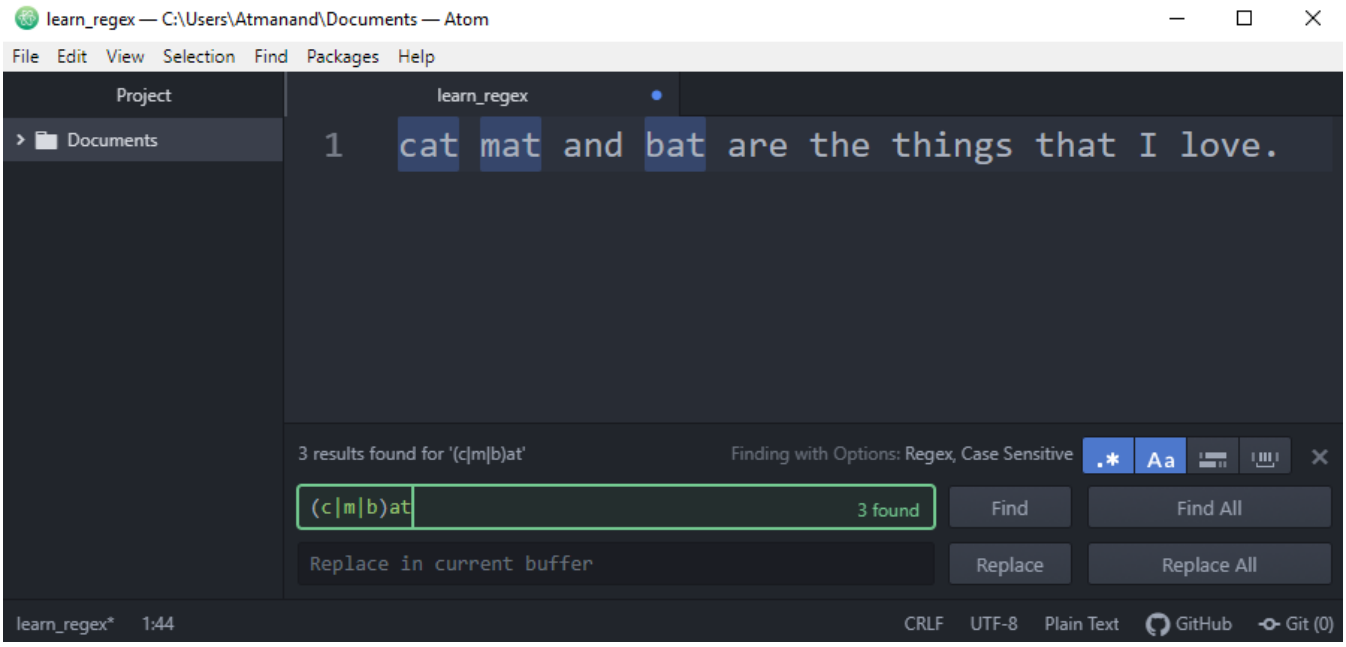
Input string:

Regular Expression:

```
(c|m|b)at
```

Matched text:

```
cat
mat
bat
```



OR Operator demonstration

Explanation:

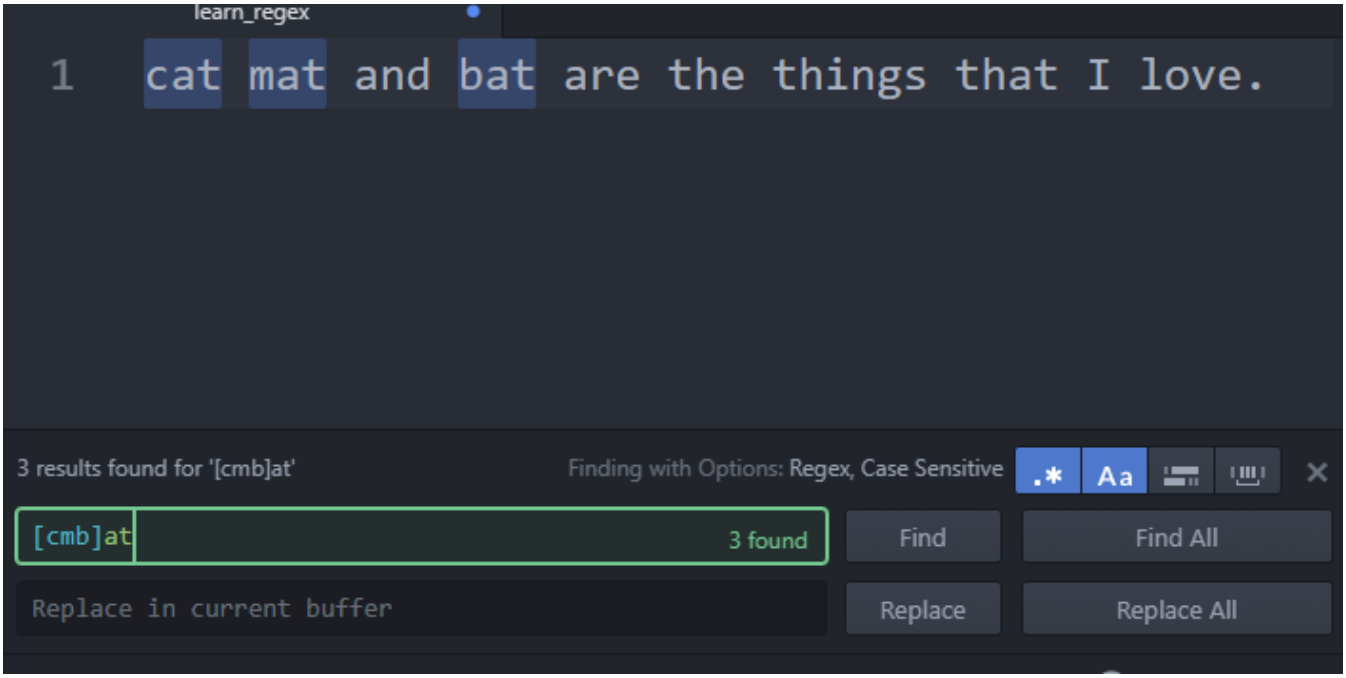
The (c|m|b) tells that the character we want to match is either c,m or b. Then the '(c|m|b)at', here 'at' is used after the OR operation. This tells that, it will match either 'cat' or 'mat' or 'bat'.

We can get the same result by using the [ ] characters. We can specify the characters we want to match inside these brackets and it will match either of them.

```
[cmb]at
```

Matched Text:

```
cat
mat
bat
```



Using square brackets to perform OR operation

. . .

3. Ranging and Bracket Expressions — [a-z, etc....]

Ranging and bracket expressions are pretty useful features of regex. They can be used to match a range of characters or match characters except for given characters.

Usage:

```
[a-z] -> Match all characters from a to z (small letters)
[A-Z] -> Match all characters from A-Z (capital letters)
```

```
[D-M] -> Match any capital letter between D and M
[2-7] -> Match any number between 2 and 7

Combining expressions:
[a-zA-Z] -> Match all letters (any small or capital letter)
[a-zA-Z0-9] -> Match all letters and numbers (any small or capital
letter or number)
```

You can also match any character EXCEPT for a single or range of characters.

Usage:

```
[^a-z] -> Match anything that is not between a to z(small letters) =
No small letters
[^A-Z] -> Match anything that is not between A to Z (capital
letters) = No capital letters
[^0-9] -> Match anything that is not between 0-9 = No numbers

[^e-v] -> Match anything that is not between e and v

Combining expressions:
[^a-zA-Z] -> No alphabets = Match only numbers
[^a-z0-9] -> Match anything except small letters and numbers = Match
only capital letters and other characters.
```

Input Text:

```
Nice song.
```

Pattern to Match:

```
i
c
e
s
o
n
g
```

```
[a-z]
```

Using Bracket Expressions to match a range of characters

Now, let's try to match some capital letters in the input string.

Input Text:

```
The name of this river is: Ganga.
```

Pattern to Match:

```
T
G
```

Regular Expression:

```
[A-Z]
```

Demonstration for the usage of the bracket expressions to match all capital letters in the input string.

. . .

4. Quantifiers

Quantifiers are regular expressions metacharacters which can be used to specify how many instances of groups, characters, bracket expressions, character ranges, etc. must be present in the input string.

List of Quantifiers:

```
yes* -> Match the string which has 'ye' followed by zero or more 's'

yes+ -> Matches the string which has 'ye' followed by one or more
's'

yes? -> Matches the string which has 'yes' or 'ye'. The character
's' is optional. It wasn't

yes{3} -> Matches the string which has 'ye' followed by exactly 3
's'

yes{2,} -> Matches the string which has 'ye' followed by 2 or more
's'
```

```
(ye)+s -> Matches the string which has one or more 'ye' and at end
's'
```

The * quantifier — Kleene Star:

Kleene Star is a quantifier used to match zero or more characters, groups, character classes, etc. in the input string. It is one of the most important quantifiers in pattern matching and parser design.

Usage:

```
yeah* -> Matches any string 'yea' which is followed by zero of more
'h'

y(eah)* -> Matches any string 'y' which is followed by zero or more
occurances of group 'eah'.

[a-zA-W]* -> Matches any string which contains any alphabet.
```

Input String:

```
yeahhhaaaboi! This is just a testing yeah yeahhhhh string to match
yah string.
```

Regular Expression:

```
yeah*
```

Patterns matched:

```
Match 1: yeahhh
Match 2: yeah
Match 3: yeahhhhh
```

Demonstration for the Kleene star quantifier.

Input String:

```
He was shouting: hellyeahyeahyeah man! I really like thing regular
expression thing! hellyeahyeah! hell!
```

Regular Expression:

```
hell(yeah)*
```

Pattern Matched:

```
Match 1: hellyeahyeahyeah
Match 2: hellyeahyeah
Match 3: hell
```

RegexBuddy Software: Demonstration of the usage of Kleene Star quantifier with groups;

The + quantifier — Kleene Plus:

Kleene Star is a quantifier used to match zero or more characters, groups, character classes, etc. in the input string. It is one of the most important quantifiers in pattern matching and writing input validators on the client as well as the server side of the Web APIs.

Usage:

```
hello+ -> Matches the string which contains 'hell' followed by 1 or
more 'o'. Example: hellooooooo, hellooo, hellooooo, hello, etc.

hel(l)*o -> Matches the string which contains 'hel' followed by 0 or
more 'l' and then followed by 'o'.

[a-zA-Z0-9_.]+ _> Matches the string contains any alphabet or number
```

Input String:

```
hellooooo! This is an example of speech synthesis in English. Well,
hellooooooo boii!!! Hi and hello.
```

Regular Expression:

```
hello+
```

Pattern Matched:

```
Match 3: hello
```


Usage of the Kleene Plus quantifier to match string which has one or more 'o's.

Input String:

```
website: www.google.com website is nice
www.facebook.com is a nice website
link: www.google.co.in temporary testing
```

Regular Expression:

```
www.[a-z]+\.(\.?[a-z]+)+
```

Explanation:

This regular expression will match a website hostname.
Starting with 'www.' , after that, there must be the name of the website like: 'google', 'facebook', etc. This name is matched by the regex [a-z]+. This will match one or more small letters from a to z. Then, it will match the group which contains a regex pattern

‘.’ is a metacharacter in the Regular Expressions(It is used to match ANY character). Then, match [a-z]+ which will match top-level domain like ‘com’, ‘org’, ‘tech’, etc. This cycle will repeat until there is no top-level domain or second-level domain left. Then, after the regex engine detects the ‘ ‘(space character), it will stop matching. And, we will successfully have our pattern matched!

Matched Pattern:

```
Match 1: www.google.com
Match 2: www.facebook.com
Match 3: www.google.co.in
```

Basic usage of Kleene Plus(+) quantifier to match the web address of a website in the input string.

The ? quantifier:

The ? quantifier is a regular expression metacharacter which is used to match the preceding character, group or character class zero or 1 time.

Usage:

```
wel(come)? -> Matches a string containing 'welcome' or 'wel'.
Because the whole group 'come' is optional.
```

Input String:

```
boy are boys! he is a boy.
That is a group of boys.
```

Regular Expression:

```
boys?
```

Explanation:

This regular expression will match the string which contains the word ‘boy’ which is followed by zero or one instance of the letter ‘s’.

Matched Pattern:

```
Match 1: boy
Match 2: boys
Match 3: boy
Match 4: boys
```

Usage of '?' quantifier to make a character, group, etc. to make that character or group optional.

Using { n }, {n, } or {m, n}:

Quantifier {n}, {n,} and {m,n} are used to specify how many instances of the preceding characters, groups or characters are to be matched. Here m and n are two whole number integers.

{n} quantifier:

This quantifier is used to match exactly ***'n'*** number of groups, characters or character classes in the input string.

Usage:

```
his{3}
```

Explanation:

Here, this regular expression will match the string 'hisss'. Because the regex engine will first find the string 'hi' and then search for 3 's' i.e. 'sss' right after that.

Input String:

```
The snake doesn't make the sound 'hiss'. It just goes like: 'hisss'
and 'hisss'.
```

Regular Expression:

```
his{3}
```

Pattern Matched:

Usage of { } quantifiers to match exactly 3 instances of 's' after 'hi'

{n,} quantifier:

{n,} quantifier is used to match 'n' or more than 'n' instances of the preceding character, group or character class.

Usage:

```
anything{3,}
```

Explanation:

This regular expression will match the string 'anythin' and then it will search for '3' or more instances of the letter 'g'. So, possibilities are it might match:

```
anythinggg
anythingggg
anythingggg
anythinggggg
anythinggg.....any number of g's
```

```
Snake is like: hisss bro! Chill the hissss out bro!
Check this hisssss out man!
```

Regular Expression:

```
his{3,}
```

Pattern Matched:

```
hisss
hissss
hisssss
```

Usage of {n,} operator for matching ‘n’ or more instances of character, character class or a group.

{m,n} quantifier:

{m,n} quantifier can be used to match the preceding characters from m,n times. Not more, not less. Here, m and n are positive integers. Here’s its usage:

```
thing{1,5}
```

Explanation:

Here, the regex engine will match the string ‘thin’ first and then search for ‘1’ to ‘5’ instances of the letter ‘g’. If it is found, the match will be found and returned.

Input String:

```
Here's the thing. The thingg is thingggg. Damn! The thinggggg goes
skkk...
```

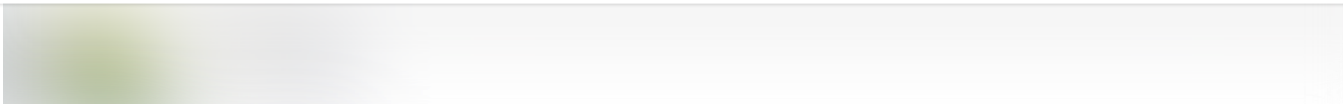
Regular Expression:

```
thing{1,5}
```

Matched Pattern:

```
thing
thingg
thingggg
thinggggg
```

Here, the string ‘thinggggg’ won’t be matched and only the part of it will be matched i.e. only ‘thingggg’ will be matched. Because in our regular expression, we only match the string which contains 1 to 5 ‘g’s.



Usage of {m,n} to match m to n instances of the preceding character, group, bracket expression or character class.

. . .

5. Character Classes

Character classes are specially defined regular expression characters which can be used to match a different kind of things in an input string. They can be used to match alphabets, numbers, special characters, etc.

Character Classes

```
\d -> Matches any number.

\w -> Matches any alphabet or number. Same behaviour as: [a-zA-Z0-0]
bracket expression.

\s -> Matches any whitespace character. This also includes matching
\r:Carriage Return, \n:new line(line feed), \t:(horizontal tab)
characters.

. -> Matches anything. Any alphabet, number, character... anything.
```

Using \d:

\d just matches any number

Input String:

```
This is my number: 1234567890. Keep it safe! :)
This another number: 9876543210. Keep it safe too! :)
```

Regular Expression:

We used + quantifier here, to match 1 or more numbers one after the other.

Matched Pattern:

```
Match 1: 1234567890
Match 2: 9876543210
```

Using \d character class to match numbers

The INVERSE of \d: ' \D '

\D is the opposite of what \d does. It matches everything else, except for numbers. It will match alphabets, symbols, etc.

. . .

Using \w:

\w matches any alphabet or number.

Input String:

Regular Expression:

```
\w+
```

Matched pattern:

```
Match 1: This
Match 2: is
Match 3: an
Match 4: example
Match 5: of
Match 6: a
Match 7: sentence
Match 8: l337
Match 9: m4n
```


Usage of \w character class to match any alphabet or number

The INVERSE of \w: '\W'

. . .

Using \s:

\s matches any whitespace, tab(\t), carriage return(\r), newline(\n), vertical tab, form feed character.

Now, we will write a basic(of course very basic) regular expression to match sentences.

Input String:

```
This is just a sentence. This is the second sentence.
And, this is the third sentence. And the fourth one.
```

Regular Expression:

```
\w+[,;\w\s]+\.
```

By writing \w+ in the beginning, we can ensure that the sentence which we are trying to match is starting with a This regular expression will perform an OR operation between ,;\w and \s. So, this will match any of these. And then, the + character will match one or more of those symbols or alphabets or whitespaces. Sound difficult at first, but it is actually pretty straightforward and simple. And finally, a full stop(.) character is matched. This will ensure that a full sentence is matched. Any characters, symbols, numbers, spaces, will be matched until there is a full stop.

Matched Pattern:

```
Match 1: This is just a sentence.
Match 2: This is the second sentence.
Match 3: And, this is the third sentence.
Match 4: And the fourth one.
```

Usage of \s to match any whitespace, tab, newline, carriage return, form feed characters.

The INVERSE OF \s: \S

\S is the opposite of what \s does. It matches everything else, except for whitespaces, tabs, newlines, etc. It will match the alphabets, numbers, etc.

. . .

Using '.' :

'. ' matches ANY character. It can be called as a wildcard. It matches anything like symbol, alphabet, number, etc. It can be used effectively with the quantifiers like + or *.

Input String:

```
[NAME]James Rolfe[/NAME]
[NAME]John Wick[/NAME]
```

Now, we want to match the name which is enclosed between those [NAME] and [/NAME] brackets. So, we can do as follows:

Regular Expression:

Here, we used \ before **[** to specify we want to match the 'actual' bracket in our input string and we don't want that bracket to act as an opening to a bracket expression or OR condition. And, then we typed out the NAME and again closed that bracket. Then, we used the '. ' character class to match anything that comes next. Then, we close the name by adding another bracket \[NAME\] to close the tag.

> ***Note** that, we have better and much safer ways to match names. We can use the '[\w.]+' expression instead of simple '.+' to ensure that only alphabets and digits are entered. But, just for the sake of simplicity, let's just ignore that fact for the time being.*

Matched Pattern:

```
Match 1: [NAME]James Rolfe[/NAME]
Match 2: [NAME]John Wick[/NAME]
```

Using the '.' character to match any character in the input string.

> ***Important Note:** If you want to match a full-stop(dot .) character in your input string, then you can '**escape**' the dot character by adding backslash(\) before the '. ' character. Escaping is the important thing to do if you want to match a character which is a reserved metacharacter in regular expression.*

. . .

6. Anchors ^ and $ (Start and end)

The anchors are basically used to define that tells the start, the end, or the both, of a line. The '^' symbol can be used to define the pattern which describes what the beginning of the line should be. And the **'$'** symbol can be used to define the pattern which describes what the end of the line should be. Let's take a look at this theory with some examples.

The '^' line start character:

The ^ character can be used with a regex pattern after it, to define how a line is starting. Or in simple words, it matches the string if the starting of the line is matched.

Input String:

```
EXCEPTION: An unhandled exception occured in the framework.\
LOG: Restarting system services...
LOG: Service restarted! All systems are active.
EXCEPTION: Host not found exception was thrown.
EXCEPTION: File not found.
```

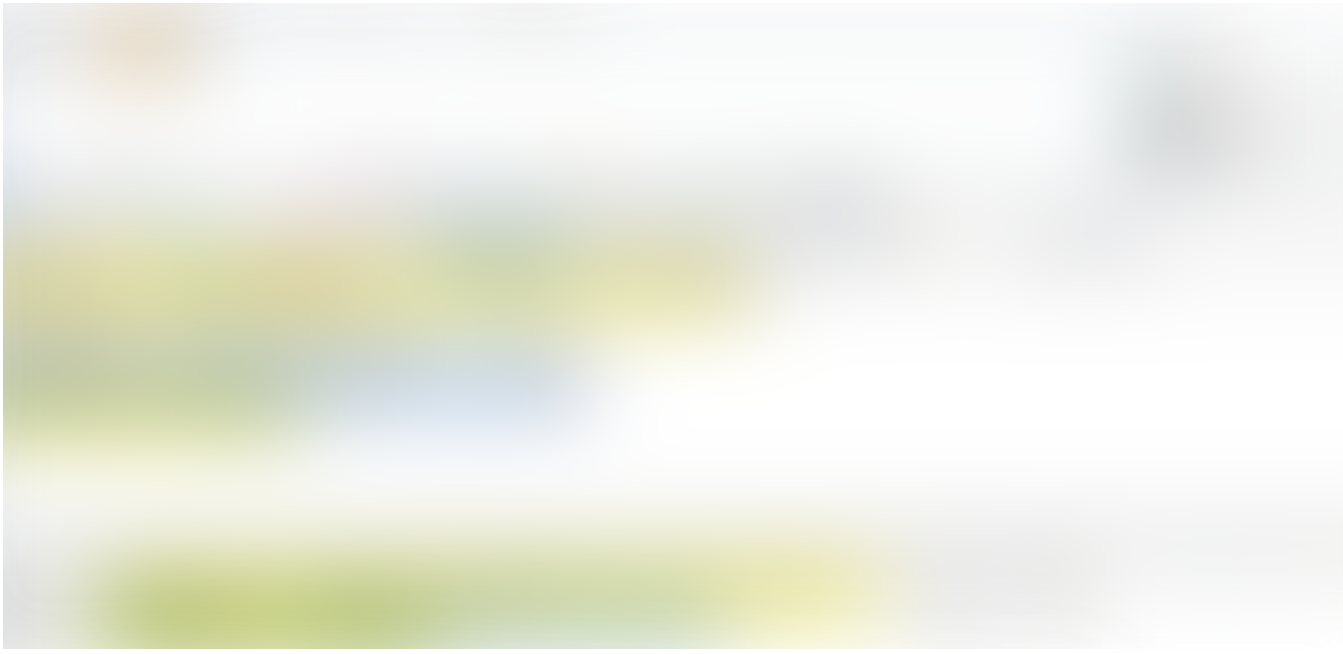
We want to match only the exceptions in this input string. So, we can use the '^' character to match the starting of the string. So, we can match the string starting with the word 'EXCEPTION'. So, the regular expression will be as follows:

Regular Expression:

```
^EXCEPTION:[\s\w]+
```

This regular expression will match the string where the **line starts** with **'EXCEPTION:'** and then match the rest of the string which contains one or more whitespace character, alphabet or any number.

```
Match 1: EXCEPTION: An unhandled exception occured in the framework
Match 2: EXCEPTION: Host not found exception was thrown
Match 3: EXCEPTION: File not found
```



Using the '^' line start anchor to match the pattern which describes how the start of the line is.

The '$' line end character:

The '$' character can be used to match the end of the line. It can be used to specify what is the line ending or in which pattern does it end.

Input String:

```
Saturday's task: Successful!
Monday's task: Failed!
Yesterday's Task: Failed!
Today's Task: Successful!
Morning's Task: Successful! Still won't match, because successful is
not at the line end.
```

So, we want to get the list of all tasks which were successful. So, we can write a regular expression as follows:

Regular Expression:

This regular expression pattern firstly matches any letter, number or symbol at first and then it matches the word ‘Successful!’. After that, it checks whether or not the word ‘Successful!’ is at the of the line. If it’s not, the pattern is not matched and it starts looking for it from the next line.

Matched Pattern:

```
Match 1: Saturday's task: Successful!
Match 2: Today's Task: Successful!
```

Using the line end ’$’ anchor to check whether or not the line ends with ’Successful!’.

Using ^ and $ combined

^ and $ can be used combined to define the pattern which describes the beginning and the ending of a line and how it should be.

Input String:

```
BEGIN This is a sample sentence END
This is not a valid sentence
```

We want to match all the lines which start by ‘BEGIN’ and end by ‘END’. So, we can write a regular expression as follows:

Regular Expression:

```
^BEGIN[ \w]+END$
```

Here, the line must start from ‘BEGIN’ and then the regex engine will search for any whitespace, any alphabet or digit. A space before ‘\w’ is on purpose. It is for performing OR operation, or simply, to match either space or a \w(alphabet or number). And, then it will analyze whether or not the line ends with ‘END’. If it does, the pattern will be matched; if not, the pattern won’t be matched.

Matched Pattern:

```
Match 1: BEGIN This is a sample sentence END
Match 2: BEGIN This is a valid sentence END
```

. . .

7. Grouping and Capturing

Grouping and capturing are one of the most important concepts of regular expressions. They are one of the most essential tools to structure the pattern matching result and make groups of the pattern matched. Later on, those groups can be reused for processing or for replacement operations. The groups are like separate match results.

If we are working on a website back-end validation system, we have to validate the form inputs before we perform any kind of operations. If we want to accept E-Mail addresses of a specific domain or if we want to take in only accepted URLs in the website link input box, we can use the regular expression grouping concept to do the job.

Captured groups are automatically named numerically according to the group number. We can also manually name them by using a specific syntax which we'll see in further sections.

Usage:

Input String:

```
www.google.com
www.facebook.com
www.some-website.com
www.something_there.com
```

Usage:

```
www.(regex_pattern).com
```

where regex_pattern is a regular expression pattern whose result will be saved in the group.

Regular Expression:

```
www.([\w_-]+).com
```

In this example, there is one group i.e. Group 1. It is automatically named since we haven't explicitly named it. We can write a regular expression inside it and whatever is matched by that regular expression will be stored in our Group 1. And, the total result will be saved in our match result.

Matched Pattern:

```
Match 1: www.google.com
 Group 1: google
```

```
Match 3: www.some-website.com
 Group 1: some-website

Match 4: www.something_there.com
 Group 1: something_there
```

So, as you can see from the matched pattern and the screenshot, in every match result, there is a captured group automatically named Group 1. This group contains the domain name of the website. Apparently, grouping and capturing is a pretty powerful tool in pattern matching and can be used flexibly to put special results into groups and reuse those groups later. In programming languages like C++, JavaScript, etc. the match and the groups are saved in an array. So, it is easy to access them and reuse them for later analytical or processing purposes.

Input String:

```
https://www.google.co.in is a nice website
ftp://ftp.welcome.com is also a nice website
```

```
(\w+)://([\w.]+)
```

This pattern matches any alphabets or digits. This match is stored in the group. This is the first group in the regex pattern, so it is **Group 1**. Then it matches '://' in the URL. Then any letter, number, or a '.' (dot) is captured and stored in the group. This is the second group in regex pattern, so it is automatically numbered Group 2.

Matched Pattern:

```
Match 1: https://www.google.co.in
 Group 1: https
 Group 2: www.google.co.in

Match 2: ftp://ftp.welcome.com
 Group 1: ftp
 Group 2: ftp.welcome.com
```

Nested Groups numbering:

The automatic group numbering is given from left to right. If there are nested groups, then the outermost group is numbered first, then the inner groups are named. If there is nesting within the nested group, then it is numbered the same way, the outer group is numbered first and the inner group is numbered later. In nested groups, the contents of the inner group ARE INCLUDED in its parent or outer group. We'll cover all these points from an example:

Input String:

```
Number is: (200)192 168 1234
And here is another one: (102)192 168 1236
```

Regular Expression:

```
\((\d+)\)(\d{3} \d{3} \d{4})
```

```
Match 1: (200)192 168 1234
 Group 1: 200
 Group 2: 192 168 1234

Match 2: (102)192 168 1236
 Group 1: 102
 Group 2: 192 168 1236
```

Grouping and Capturing demonstration

Hoped this example cleared your doubts.

Non-capturing groups:

Sometimes, you want to make the regular expression more optimized, more readable and make it use less memory to store data but still, want to keep it clean. So, you can use non-capturing groups. These are the kind of groups whose captured result is not stored into memory. They are the just kind of temporary groups which can turn out helpful to structure your regular expression properly. They are helpful to manage the back-references which we're going to discuss in our last topic.

Usage:

Here, whatever the '**expression**' matches, it is not stored in a new group. It is just checked and not stored as a separate group, which is pretty beneficial in some situations.

Input String:

```
Hacker123
Programmer234
TestUser12314
hackerman3412
damnman
godsoul
cppguy456
```

Regular Expression:

```
([a-zA-Z]+)(?:\d+)
```

As we used '?:' in the group, this group is a non-capturing group and will not be captured as a separate group. This regular expression will match a simple username with some alphabets in the beginning and a number at the end. Only the alphabets part will be captured and the number won't be captured.

Matched Pattern:

```
Match 1: Hacker123
Group 1: Hacker

Match 2: Programmer234
Group 1: Programmer

Match 3: TestUser12314
Group 1: TestUser

Match 4: hackerman3412
Group 1: hackerman
```

Demonstration of non-capturing groups.

Named Capturing Groups(explicitly named groups)

Named capturing groups are those groups to which we have explicitly given a name or an alias which can be referenced or backreferenced later. Basically, the purpose of naming capturing groups is to make the groups more easily accessible. Because it's difficult to remember the automatically allocated group named like 1,2, etc. It is a nice feature and turns out to be pretty useful in some cases.

Usage:

```
(?<group_name>regular_expression)
OR
(?'named_group'regular_expression)
```

Input String:

```
His Birth date: 10-12-1990
Her Birth date: 11-12-1990
Their Birth date: 20-12-1992
```

Regular Expression:

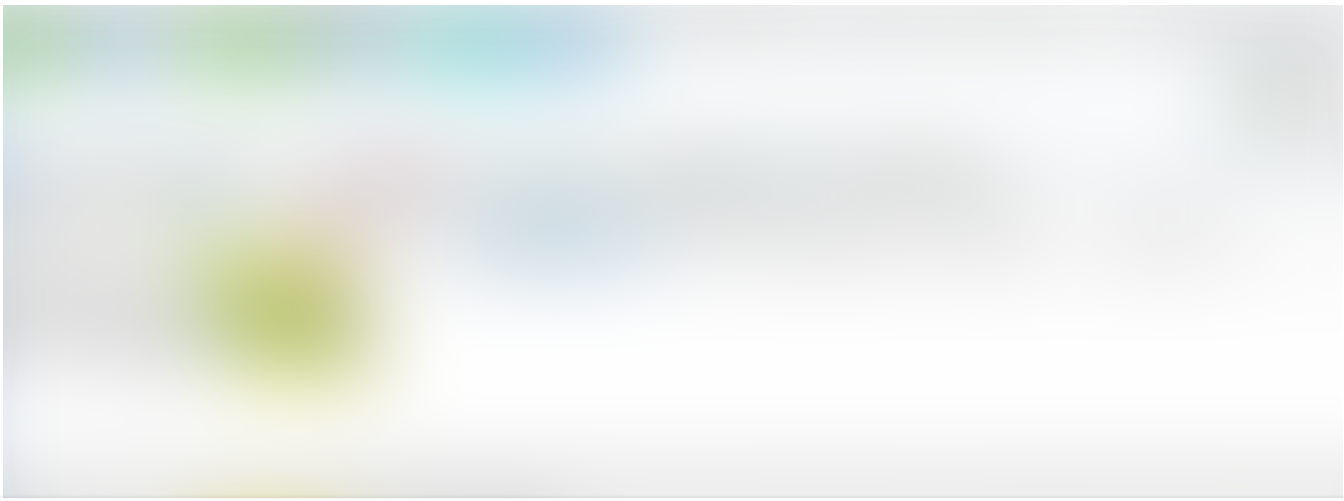
```
(?<date>\d{2})-(?<month>\d{2})-(?<year>\d{4})
```

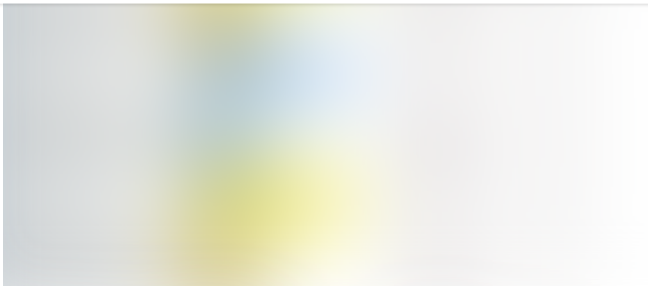
Matched Pattern:

```
Match 1: 10-12-1990
 Group "date": 10
 Group "month": 12
 Group "year": 1990

Match 2: 11-12-1990
 Group "date": 11
 Group "month": 12
 Group "year": 1990

Match 3: 20-12-1992
 Group "date": 20
 Group "month": 12
 Group "year": 1992
```





Demonstration of named capturing groups and its uses.

. . .

Word boundaries (\b)

Word boundaries are the boundaries between a word and a non-word character. In other words, it is a boundary between \w(any alphabet or number) and its opposite \W(any non-alphabet or non-number). Word boundaries can be used to match a complete word or to match everything within 2 boundaries(non-word characters).

Input String:

```
Capital Letter word.
This is another Capital Letter Word.
```

Regular Expression:

```
\b[A-Z]\w*\b
```

Matched Pattern:

```
Match 1: Capital
Match 2: Letter
Match 3: This
Match 4: Capital
Match 5: Letter
Match 6: Word
```

new word and some letters in a word. And, then we then tell regex engine to match any capital letter and then match 0 or more any letters or numbers. Then, we again add a word boundary to ensure that the string which is being matched will only be matched till a word boundary(like space)

. . .

Backreferencing

A lot of times, there is a need to go back into the regular expression and access what was previously matched and then reuse that matched sequence of characters. For instance, when parsing basic XML tags, we need to confirm that the opening tag and the closing tag have the same tag name. So, how do we do that? Well, Backreferencing provides a solution for that.

What is backreferencing?

Backreferencing is a technique and tool that can be used to reference and get stored data from the captured groups and put it in the place where backreference is requested. It is very helpful to get the value of a captured group and use or compare it later with a backreference.

Usage:

```
(captured_group)some_stuff\1
    Here, \1 will get the data of first captured group i.e.
(captured_group).

(?<name>regular_expression)...some_stuff\k<name>:
    Here, \k<name> will get the data of captured grouop named
'name'.

(?<name>regular_expression)...some_stuff\k'name'
    Here, \k'name' will be the same as \k<name>
```

Examples:

Input String:

```
<tag3>And some more stuff...</tag3>
```

Regular Expression:

```
<(\w+)>([\w\s.]+)</\1>
```

Pattern Matched:

```
Match 1: <tag1>Stuff</tag1>
Group 1: tag1
Group 2: Stuff

Match 2: <tag2>Some more stuff</tag2>
Group 1: tag2
Group 2: Some more stuff

Match 3: <tag3>And some more stuff...</tag3>
Group 1: tag3
Group 2: And some more stuff...
```

This regular expression pattern will match a simple XML tag. Here, the tag name is captured in a group[(\w+)] i.e. Group 1. And, then it is back referenced later using \1. Here, \1 will get the content of Group 1 and put it in its place. If the complete pattern is matched, we will get the result string which is a simple XML tag. The tag name will be stored in Captured Group 1 and the inner data of tag will be stored in Captured Group 2.

Example: Checking for double words.

```
Welcome to the the party of programmers.
This this is a a nice party!
```

Regular Expression:

```
\b([\w\d]+)\s+\1\b
```

This regular expression will search for a repeated word in complete text input. By entering \b at the front, we ensure that what we are searching for is a whole word starting after any non-alphabet and non-number character. And then we backreference the captured word and this will check whether the word is repeated or not. And then, again by entering a word boundary \b, we ensure that the backreferenced word occurs before a word boundary(like space)

Matched Pattern:

```
Match 1: the the
Group 1: the

Match 2: a a
Group 1: a
```


Using backreferencing and word boundaries to check search for a doubled word.

10. Greedy and Lazy Quantifiers

Greedy Quantifiers: Searches for the longest string
Lazy Quantifiers: Searches for the shortest string

Suppose we have a situation like this:

```
<name>John Wick</name><name>Robert McCall</name>
```

Regular Expression:

```
<name>(.*)</name>
```

Matched Pattern:

```
Match 1: <name>John Wick</name><name>Robert McCall</name>
Group 1: John Wick</name><name>Robert McCall
```

Now, this is disappointing. This is not what we actually wanted to match. We wanted to match the data inside the name tag and the nearest closing name tag.

Expected match:

```
Match 1: <name>John Wick</name>
Group 1: John Wick

Match 2: <name>Robert McCall</name>
Group 1: Robert McCall
```

This is because the * quantifier is **'GREEDY'**. It is **searching for the longest match**. It searches for <name> and then the longest match possible. And then our regex matches </name>. This is not what we want. We want to match the nearest closing name tag.

Lazy Quantifiers:

We can do this with the help of 'LAZY' quantifiers. Lazy quantifiers **search for the smallest match**. We can make quantifiers lazy by adding a '?' after the quantifier. Now, that quantifier will search for the smallest group possible.

```
<name>(.*?)</name>
```

Matched Pattern:

```
Match 1: <name>John Wick</name>
Group 1: John Wick

Match 2: <name>Robert McCall</name>
Group 1: Robert McCall
```

Now our regular expression is matching the correct pattern as we expected.

Usage of lazy quantifiers to match the smallest pattern possible.

Similarly, we can make other quantifiers lazy to match the smallest group. You may refer to the table given below.

Greedy and Lazy quantifiers table:

```
+------------------+-----------------+----------------------------+
| Greedy quantifier | Lazy quantifier |       Description          |
+------------------+-----------------+----------------------------+
```

```
| {n}                | {n}?             | Quantifier: exactly n       |
| {n,}               | {n,}?            | Quantifier: n or more       |
| {n,m}              | {n,m}?           | Quantifier: between n and m|
+--------------------+------------------+-----------------------------+
```

> ***Credits**: Premraj*

Simple Examples(Input String: stackoverflow):

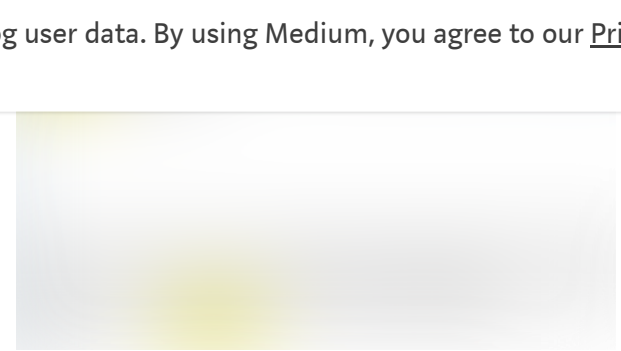
Greedy Quantifier:

```
s.+o
Matched: stackoverflo
```


Greedy quantifier

Lazy Quantifier:

```
s.+?o
Matched: stacko
```



Lazy Quantifier

> ***Credits**: Premraj*

Important Note:

Lazy quantifiers are costlier than the greedy ones if not implemented correctly. They might take more time and their complexity is higher. Same applies to greedy quantifiers. So, writing efficient expressions is important.

. . .

So, I hope this tutorial would've helped you understand the basics of regular expressions. Links to documentation, tools, and references are given below. Happy Pattern Matching! :)

Written by: Atmanand Nagpure (proghax333)

. . .

* Documentation and References:

- Regular Expressions Wiki: https://en.wikipedia.org/wiki/Regular_expression
- Regular-Expressions.info: https://www.regular-expressions.info/tutorial.html
- Interactive Learning — RegexOne: https://regexone.com/
- Ryan's Tutorials: https://ryanstutorials.net/regular-expressions-tutorial/

* Tools:

- **Atom Text Editor**: https://atom.io/

- **RegExr**: https://regexr.com/
- **Regex101**: https://regex101.com/

. . .

Atmanand Nagpure:

Medium: https://medium.com/@skillzworldtech
Blogger: https://www.skillzworldtech.ml

. . .

Regex Regular Expressions Programming Text Processing Pattern Matching