

---

# Regular Expressions From Beginner To Expert By Varun

Date: November 12, 2018

In this document want to show how to master regex. I wasn't good at regex until I realized its power and the necessity to understand and master it.

---- Avoid writing lines and lines of code by using power of regex.

---- The regex shown here will take you from a simple example to the complex patterns.

---- Lot of tools apart from all the programming languages support regex.

---- I personally know integration tools such as **Ca-API-Gateway**, **Apigee**, **TibcoFlogo**, **Mulesoft** where regex can be used and match complex patterns, save unnecessary code for pattern match.

---- Logging slutions like **ELK**, **Splunk** support full on regex.

---- Analytical tools such as **Tableau** support full on regex.

---- Please follow ths document and practice along by the end of it. You will start writing complex regex on your own.

## Contents

<b>1.1</b>	<b>Introduction .....</b>	<b>3</b>
1.1.1	What are regular expressions.....	3
1.1.2	Why to learn regular expressions .....	3
1.1.3	How regex is created .....	3
1.1.4	Tools Used .....	4
1.1.5	Basic Syntax Regex .....	5
<b>1.2</b>	<b>All About Characters .....</b>	<b>5</b>
1.2.1	Character Literals.....	5
1.2.2	Character Classes .....	6
1.2.3	Forward Boundary .....	7
1.2.4	Character Range.....	8
1.2.5	Regex Live Example .....	10
1.2.6	Negation Characters .....	12
<b>1.3</b>	<b>Meta Characters .....</b>	<b>13</b>
1.3.1	What are Meta Characters .....	13
1.3.2	Wild Card Meta Characters Part 1 .....	15
1.3.3	Escaping Meta Characters .....	19
1.3.4	Predifined Character Classes .....	20
<b>1.4</b>	<b>Anchors And Word Boundary.....</b>	<b>22</b>
1.4.1	Anchors.....	22
1.4.2	Word Boundary .....	24
<b>1.5</b>	<b>Quantifiers .....</b>	<b>26</b>
1.5.1	? quantifier .....	26
1.5.2	* Quantifier .....	28
1.5.3	Matching Example.....	28
1.5.4	+ Quantifier .....	32
1.5.5	{ } Quantifiers To Limit Range .....	33
1.5.6	{min,} Quantifiers .....	33
1.5.7	{min,max} Quantifiers .....	34
1.5.8	Greedy Quantifiers .....	35

1.5.9	Lazy Or Reluctant Quantifiers.....	36
1.5.10	Alternative For Lazy Quantifiers.....	36
1.5.11	Greedy Quantifiers vs Lazy Quantifiers.....	38
1.5.12	Increasing the performance of the RegeX.....	39
1.6	Groups In Regex .....	40
1.7	Assertions.....	40
1.8	Real Life Scenarios .....	40

## 1.1 Introduction

### 1.1.1 What are regular expressions

- Regular expressions are also called as Regex or Regexp
- Sequence of characters that defines a search pattern

### 1.1.2 Why to learn regular expressions

Used in all programming languages. Also used in caApiGateway and Apigee

- Can be used to validate a form
- To extract a part of string
- To clean up a file
- used in database or command line as well
- used for other multiple purposes and list is on and on.

For example validating user input into a form

First name	Last name
<input type="text" value="Sandeep"/>	<input type="text" value="Kumar"/>

Username

Sorry, only letters (a-z), numbers (0-9), and periods (.) are allowed.

Password <input type="password"/>	Confirm password <input type="password"/>	
--------------------------------------	--	---

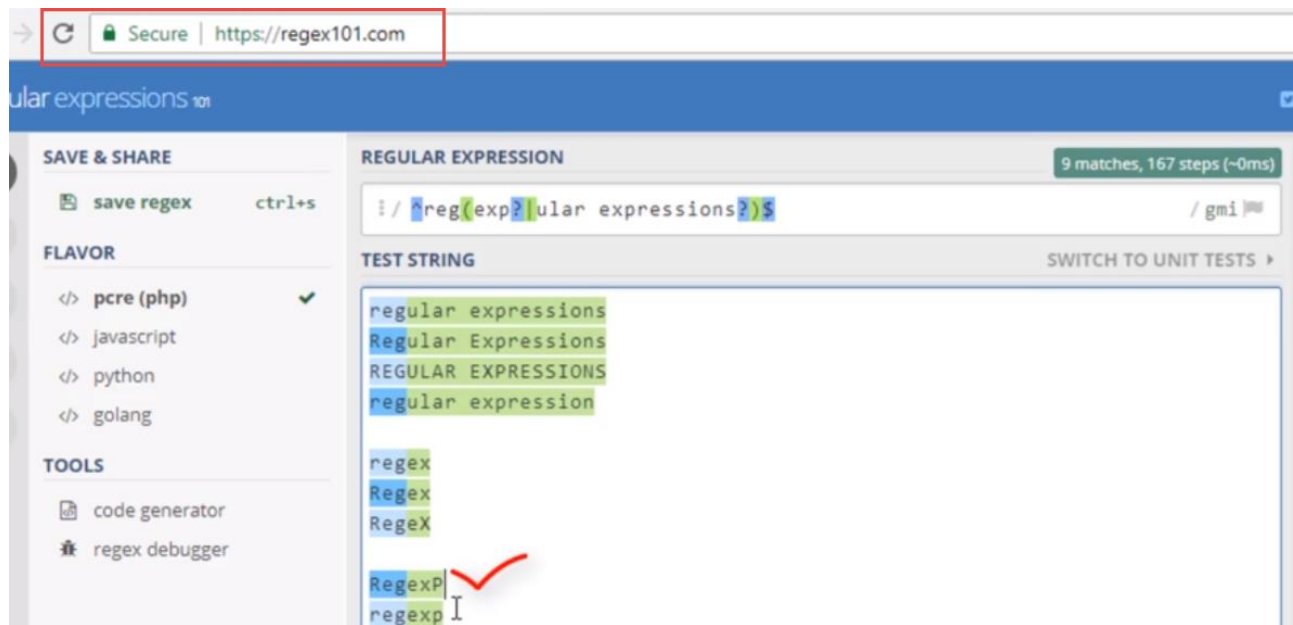
Please choose a stronger password. Try a mix of letters, numbers, and symbols.

### 1.1.3 How regex is created

# /^reg(exp?|ular expressions?)/gmi



One simple example of a Regular Expression



## 1.1.4 Tools Used

Tools used for learning the regex during the course of this document.

### Online regex tester

<https://regex101.com>

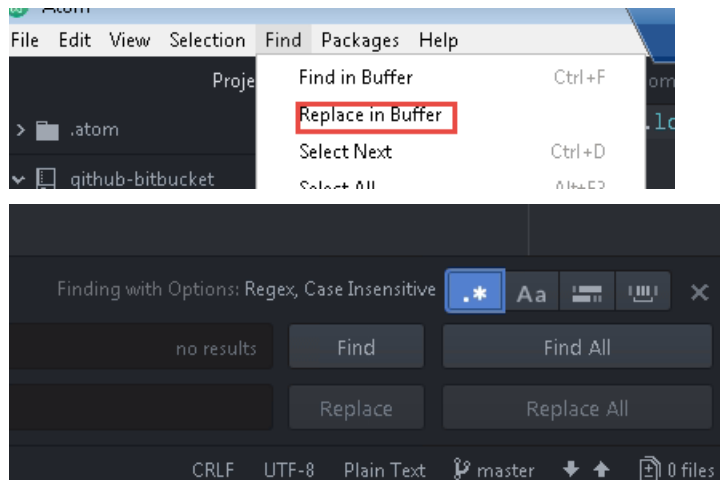
<https://regexr.com>

<https://www.regextester.com>

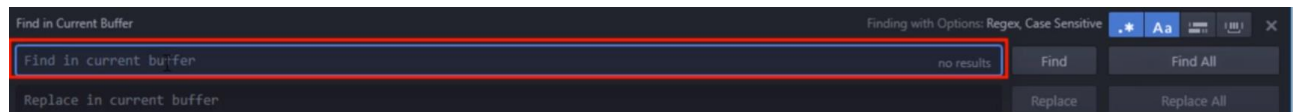
### Offline Regex Testers

<https://atom.io>

reach regex by using ctrl+F or below steps

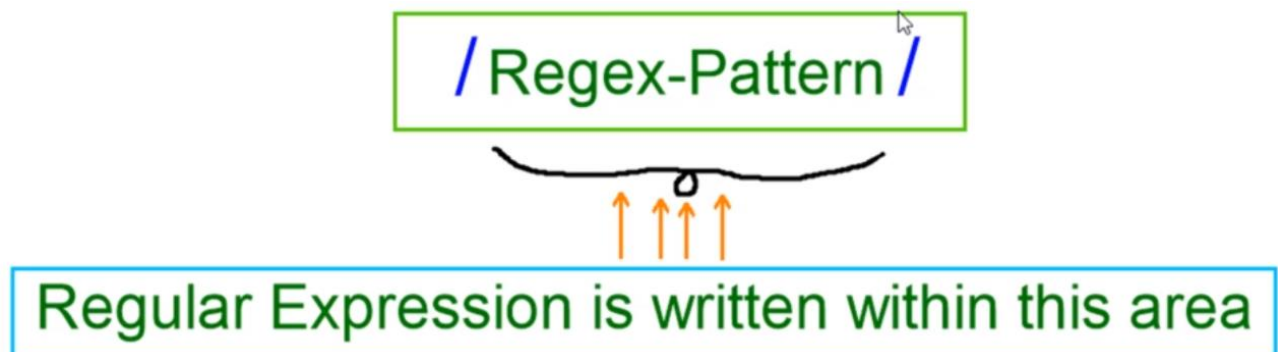


Regexp are case sensitive.



### 1.1.5 Basic Syntax Regex

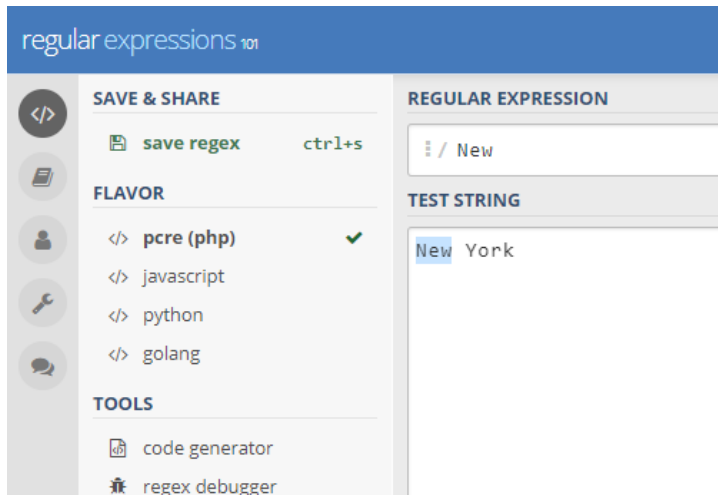
Used in all programming languages. Also used in Api Management tools such as, caApiGateway and Apigee



## 1.2 All About Characters

### 1.2.1 Character Literals

Takes the string as it is.



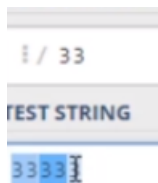
Regexby default is case sensitive.

:/Java/gm (Case sensitive)

:/Java/gmi (Case insensitive)

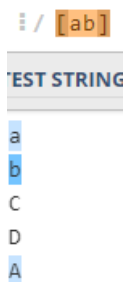


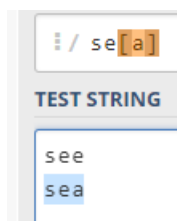
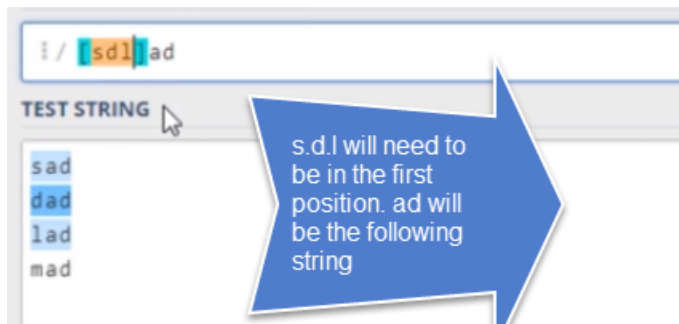
Regex starts searching from left to right



### 1.2.2 Character Classes

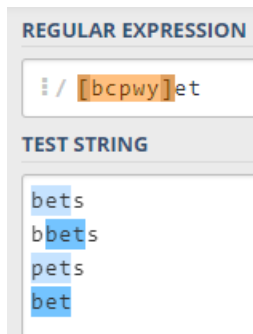
Character class are enclosed in [] brackets





### 1.2.3 Forward Boundary

>> Forward boundary uses the following pattern `'\b'`



say in this you want to match only the word bet and not sub string that may contain bet.



**REGULAR EXPRESSION**

```
// \b[bcpsy]et|
```

**TEST STRING**

```
bets
bbets
pets
bet
```

**REGULAR EXPRESSION**

```
// \b[bcpsy]et\b
```

**TEST STRING**

```
bets
bbets
pets
bet
pet
wet
```

#### 1.2.4 Character Range

**EXPLANATION**

- ▼ `[a-z0-9]` / `gm`
  - ▼ **Match a single character present in the list below** `[a-z0-9]`
    - `a-z` a single character in the range between `a` (index 97) and `z` (index 122) (case sensitive)
    - `0-9` a single character in the range between `0` (index 48) and `9` (index 57) (case sensitive)
  - ▼ **Global pattern flags**
    - `g` **modifier**: global. All matches (don't return after first match)
    - `m` **modifier**: multi line. Causes `^` and `$` to match the begin/end of each line (not only begin/end of string)

**QUICK REFERENCE**

- all tokens
- ★ **common tokens** ✓
- general tokens
- anchors
- meta sequences
- \* quantifiers
- group constructs
- character classes

A single character of: a, b or c	<code>[abc]</code>
A character except: a, b or c	<code>[^abc]</code>
A character in the range: a-z	<code>[a-z]</code>
A character not in the range: a-z	<code>[^a-z]</code>
A character in the range: a-z or A-Z	<code>[a-zA-Z]</code>
Any single character	<code>.</code>
Any whitespace character	<code>\s</code>
Any non-whitespace character	<code>\S</code>
Any digit	<code>\d</code>

REGULAR EXPRESSION

/ [a-zA-Z0-9]

TEST STRING

my name is ABCDEFGHdolf  
 my name is adolf  
 my name is adolf ddb 12345

Say there are 5 employees and you want to match only the emp 1,2,3

REGULAR EXPRESSION

/ emp[123]

TEST STRING

emp1  
 emp2  
 emp3  
 emp4  
 emp5

Write a regex to match the first 3 words.

REGULAR EXPRESSION

/ insert your r

TEST STRING

Agx  
 Bhy  
 Ciz  
 =====  
 lklkjj  
 lkjl  
 uiih  
 sdfad

REGULAR EXPRESSION

/ [A-C][g-i][x-z]

TEST STRING

Agx  
 Bhy  
 Ciz  
 =====  
 lklkjj  
 lkjl  
 uiih  
 sdfad

Indexes of alphabets and numbers

A	-	65
B	-	66
..		
Z	-	90
=====		
a	-	97
b	-	98
...		
z	-	122
=====		
0	-	48
1	-	49
..		
9	-	57

You need to match the index when defining the character ranges you need to go from small to big

[A-z] This is correct

[a-Z] This is wrong.

### 1.2.5 Regex Live Example

```
<!DOCTYPE html>
<html>
<head>
<title>Testing Regular Expressions in HTML forms</title>
</head>
<body>
<h1>Test your Regular Expressions here</h1>
<fieldset>
<legend>Write your Regular expressions</legend>
<form>
<div>
<label for="display-name"> Prove that you are not a robot:
<span class="warning">*(Enter a single English capital letter.)</span>
</label>
<input type="text" id="display-name" name="ip-display"
pattern="[A-Z]"
title="Enter any single English capital Letter only"/>
</div>
<div>
<input type="submit" class="submit" value="Submit" />
</div>
</form>
</fieldset>
</body>
</html>
```

```

<!DOCTYPE html>
<html>

<head>
  <title>Testing Regular Expressions in HTML forms</title>
</head>

<body>
  <h1>Test your Regular Expressions here</h1>

  <fieldset>
    <legend>Write your Regular expressions</legend>
    <form>
      <div>
        <label for="display-name"> Prove that you are not a robot:
        <span class="warning">*(Enter a single English capital letter.)</span>
        </label>
        <input type="text" id="display-name" name="ip-display"
        pattern="[A-Z]"
        title="Enter any single English capital Letter only"/>
      </div>
      <div>
        <input type="submit" value="Submit" />
      </div>
    </form>
  </fieldset>

</body>
</html>

```

This 'pattern' attribute in HTML uses Regular Expression

```

<!DOCTYPE html>
<html>

<head>
  <title>Testing Regular Expressions in HTML forms</title>
</head>

<body>
  <h1>Test your Regular Expressions here</h1>

  <fieldset>
    <legend>Write your Regular expressions</legend>
    <form>
      <div>
        <label for="display-name"> Prove that you are not a robot:
        <span class="warning">*(Enter a single English capital letter.)</span>
        </label>
        <input type="text" id="display-name" name="ip-display"
        pattern="/[A-Z]/"
        title="Enter any single English capital Letter only"/>
      </div>
      <div>
        <input type="submit" class="submit" value="Submit" />
      </div>
    </form>
  </fieldset>

</body>
</html>

```

You do not need to put forward slash at start and end when you use pattern attribute.

## Test your Regular Expressions here

Write your Regular expressions

Prove that you are not a robot: \*(Enter a single English capital letter.)

Submit

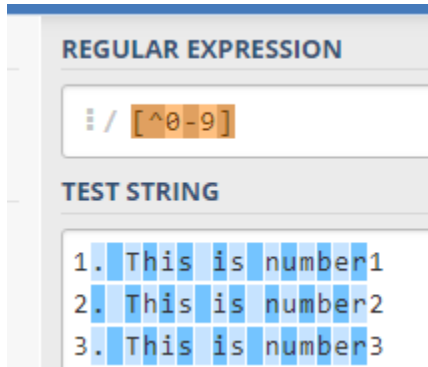


Please match the requested format.  
Enter any single English capital Letter only

### 1.2.6 Negation Characters

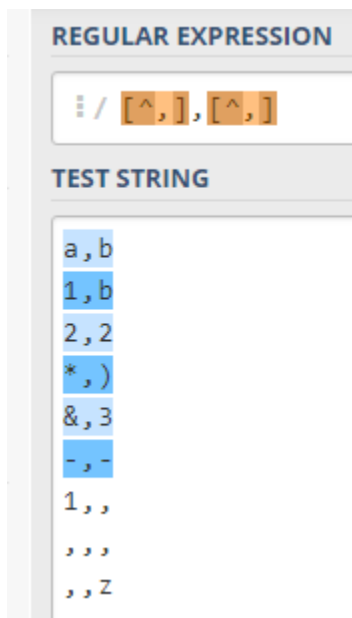
Are represented with a ^ symbol inside [] brackets.

[^a-e] will match all characters except a-e



Example 1:-

- The First Character should be any character except a comma(,)
- The Second Character must be a comma(,)
- The Third Character should be any character except a comma(,)



Example 2:-

- The First Character must be a vowel(a,e,i,o,u)
- The Second Character must be a English alphabet(it can be lowercase or uppercase English alphabets)

- The Third Character must a single number
- The Fourth Character should be any character except a Hyphen(-)
- The Fifth Character must be a Hyphen(-)
- The Sixth Character should be any character except a Hyphen(-)

The screenshot shows a web-based regular expression testing tool. It has two main sections: 'REGULAR EXPRESSION' and 'TEST STRING'. The 'REGULAR EXPRESSION' section contains the regex `/[aeiou][A-z][0-9][^-]-[^-]/`. The 'TEST STRING' section contains a list of test strings: 'eA9g-b', 'aM6.-)', 'un06-', 'eA9--o', 'aM6.--', and '8M5)))'. The first three strings are highlighted in blue, indicating they match the regex.

REGULAR EXPRESSION	TEST STRING
<code>/[aeiou][A-z][0-9][^-]-[^-]/</code>	eA9g-b
	aM6.-)
	un06-
	eA9--o
	aM6.--
	8M5)))

## 1.3 Meta Characters

### 1.3.1 What are Meta Characters

In regex each character is described as normal character or meta character.

Special character that have special meaning.

\	Backslash
^	Caret
\$	Dollar
.	Period or Dot
?	Question mark
*	Star or Asterisk
+	Plus symbol
{ }	Curly brackets
	Pipe symbol
( )	Parenthesis
[ ]	Square brackets
=	Equal to
!	Exclamation mark

## Meta characters

\ used to precede a meta char or predefine character class.

^ negation character and used in anchors

\$ Is used for matching a boundary at the end of the line

. used to match any character apart from new line

? quantifiers

\* quantifiers

+ quantifiers

{ } quantifiers

| Alternation

( ) Grouping

[ ] Ranges

= Assertions (look ahead, look behind)

! Assertions (look ahead, look behind)

### 1.3.2 Wild Card Meta Characters Part 1

. meta character is also called wild card metacharacter



To match only one character



**REGULAR EXPRESSION**

:/ \b.\b

**TEST STRING**

```
java
php

Mainframe
Python
Perl

Ruby
  Javascript
    Html
a[
cat
```

New line char is represented by \n. Line break.

**REGULAR EXPRESSION**

:/ \n\na

**TEST STRING**

```
eA9g-b
aM6.-)
  g
un06-,
eA9--o
aM6.--
8M5)))
```

\s whitespace

Matches new line followed by whitespace

REGULAR EXPRESSION

/ \n\s

TEST STRING

eA9g-b  
aM6.-)  
g  
un06-  
eA9--o  
  
aM6.--  
8M5))

\b is a boundary between a word and a non word character

REGULAR EXPRESSION

/ \b.re

TEST STRING

are  
\*re  
kre  
%re  
hre  
&re  
ore  
mre  
lre  
sre  
re  
  
re  
  
Aare  
Bare  
Cure  
More

REGULAR EXPRESSION

/ ba.

TEST STRING

bad  
bag  
ba\*  
ba#  
ba  
baz  
bam  
ba  
bat  
bay

REGULAR EXPRESSION

/ [A-Za-z]. [A-Za-z]

TEST STRING

Mom  
M#m  
M(m  
Dad  
D.d  
D1d  
Bob  
b9b  
Gag  
g&h  
Pop  
P\$p  
Tit  
T%t  
Tat  
t@t  
WOW  
W~W

**REGULAR EXPRESSION**`/. [A-Z][a-z][0-9].`**TEST STRING**

```
1235
%Gk9.
    Pol_
1pM10
1111
lkjl
Michael -
Hi%Ak0.How are you?
Sam - 1pM10
care
love
{}-{}
^
*7^5
```

- 1) Character can be anything other than a newLine
- 2) Any upper case
- 3) ANY lowercase
- 4) Number
- 5) Any character other than new line

**REGULAR EXPRESSION**

```
^.[A-Z][a-z]\d[^\n]
```

**TEST STRING**

```
12345
%Gk9.
Pol_
1pM10
1111
lkjl
Michael
Hi%Ako.How aree you?
Sam - 1pM10
```

Alternate approach

Using Predefined Character classes

### 1.3.3 Escaping Meta Characters

Meta characters are escaped by \

REGULAR EXPRESSION	REGULAR EXPRESSION
<code>\\</code>	<code>\+</code>
TEST STRING	TEST STRING
<code>\^\$.?*\+ ()[]</code>	<code>\^\$.?*\+ ()[]</code>

## 1.3.4 Predifined Character Classes

## Predefined Character Classes List-1

**\d** Matches only numbers  
**\D** Matches everything apart from numbers

**\w** Matches only word characters [A-Za-z0-9\_] [A-Za-z0-9\_]  
**\W** Matches non word characters

**\s** Matches whitespace.  
**\S** Matches nonwhitespace

REGULAR EXPRESSION	REGULAR EXPRESSION	REGULAR EXPRESSION	REGULAR EXPRESSION
<code>// \w</code>	<code>// \w</code>	<code>// \s</code>	<code>// \S</code>
TEST STRING	TEST STRING	TEST STRING	TEST STRING
Varun Is A Good_Boy	<code>\^\$.?*[+ ()[]</code>	<code>\^\$.?*[+ ()[]</code>	<code>\^\$.?*[+ ()[]</code>

REGULAR EXPRESSION
<code>// &lt;h\d&gt;</code>
TEST STRING
<pre> &lt;!DOCTYPE html&gt; &lt;html&gt; &lt;head&gt; &lt;title&gt;Testing Regular Expressions in HTML forms &lt;/head&gt; &lt;body&gt; &lt;h1&gt;Test your Regular Expressions here&lt;/h1&gt; &lt;h2&gt;Test your Regular Expressions here&lt;/h1&gt; &lt;h3&gt;Test your Regular Expressions here&lt;/h1&gt; &lt;h4&gt;Test your Regular Expressions here&lt;/h1&gt; </pre>

To capture a 3 digit number within ""

**REGULAR EXPRESSION**

// `"\d\d\d"`

**TEST STRING**

```
<!DOCTYPE html>
<html>
<head>
<title>Test
</head>
<body>
("50000")
("800",4)
```

**REGULAR EXPRESSION**

// `"\d{3}"`

**TEST STRING**

```
<!DOCTYPE html>
<html>
<head>
<title>Testing Regular Ex
</head>
<body>
("50000")
("800",4)
<h1>Test your Regular Expressions here</h1>
<h2>Test your Regular Expressions here</h1>
<h3>Test your Regular Expressions here</h1>
<h4>Test your Regular Expressions here</h1>
```

Match all numbers with length 3 within "". The character class can be limited using {}

**REGULAR EXPRESSION**

// `\s`

**TEST STRING**

```
<!DOCTYPE html>
<html>
<body>
ch 15 *****
up 0: " before me, there is a tab
: 124-125 *****
There are 3 spaces before t
```

## Predefined Character Classes List-2

`\n` Line break or New line character

`\t` Tab character

non- printable characters  
 \r is carriage return  
 \f is form feed

**REGULAR EXPRESSION** 18 matches, 126 steps (~1ms)

`/\n</code>`

**TEST STRING** SWITCH TO UNIT TESTS

```

*****
<p>Please match all the numbers.</p>
<button onclick="func()">Try it out</button>
<html>
<p id="demo"></p>

```

**REGULAR EXPRESSION**

`/\s`

**TEST STRING**

```

<!DOCTYPE html>
<html>
<body>\n
*****
Before me, there is a tab
*****
There are 3 spaces before this
tttt

```

## 1.4 Anchors And Word Boundary

### 1.4.1 Anchors

Are used to match before the beginning or end of

- 1) String
- 2) Line

# Anchors



Used to match before the beginning of string or line



Used to match right after the end of the string or line

^

Used to match before the beginning of string or line

\$

Used to match right after the end of the string or line

When 'Caret' is used inside the square bracket, it is negation character

REGULAR EXPRESSION

no match

EXPLANATION

1/ Insert your regular expression here

TEST STRING

SWITCH TO UNIT

Matches start and end of each line

-----Original Message-----

real.com]

LHS@loreal.c

al.com>

REGEX FLAGS

global

Don't return after first match

multi line

^ and \$ match start/end of line

insensitive

Case insensitive match

extended

Ignore whitespace

eXtra

Disallow meaningless

REGULAR EXPRESSION

TEST STRING

1

<!DOCTYPE html>

REGULAR EXPRESSION

TEST STRING

-----Original Mes

From: [redacted]

Sent: [redacted]

To: [redacted]

Cc: KOT

Subject: uscorgulay7p.na.loreal.intra -API Gate

Matches all lines starting with number

REGULAR EXPRESSION

TEST STRING

Match all senteces starting with word

-----Original Message-----

From: [redacted]

Sent: [redacted]

To: [redacted]

Cc: KOT

Subject: [redacted]

The

1

Serv

1

Routing

Response body = <html>

REGULAR EXPRESSION

TEST STRING

End of line

-----Original Message-----

The following Service has an issue with a routing failure

1

Service name = LM\_SAPInterFace

Routing reason code = 504

Response body = <html>

<head><title>504 Gateway Time-out</title></head>

<body bgcolor="white">

<center><h1>504 Gateway Time-out</h1></center>



The first screenshot shows the regular expression `/He` being tested against a string. A red callout bubble points to the matches, stating: "Match all lines starting with He".

The second screenshot shows the regular expression `/day$` being tested against the same string. A red callout bubble points to the matches, stating: "All lines ending with Day".

The third screenshot shows the regular expression `/\w{3} Program: Successful` being tested against a string. A red callout bubble points to the match, stating: "Match the line starting with 'Java' and ending in 'Successful'. I have used character classes to make it look fancy and complex.. You can also use `^Java Program: Successful/gm`".

## 1.4.2 Word Boundary

Is used to match the boundary between word and non-word character.

The screenshot shows the regular expression `/\b` being tested against a string. The matches are highlighted in blue. Below the matches, a blue box contains the text: "Word Boundary".

Below the blue box, the text `/\B` is shown, followed by the text "Not a Word Boundary".

**REGULAR EXPRESSION** -377ms

`/ Successful`

**TEST STRING**

```
Comment out all Java Program: successful
Java Program: unsuccessful
Java Program: Successful
***&Java Program:successful**)
this cobol program: successful
that PHP Program: unsuccessful
```

**REGULAR EXPRESSION** mi

`/ \bSuccessful`

**TEST STRING** ESTS

```
Comment out all Java Program: successful
Java Program: unsuccessful
Java Program: Successful
***&Java Program:successful**)
this cobol program: successful
that PHP Program: unsuccessful
```

**REGULAR EXPRESSION** 3 matches, 20600 steps (~28ms)

`/ \bSuccessful\b`

**TEST STRING** SWITCH TO UNIT TESTS

```
Comment out all Java Program: successful
Java Program: unsuccessful
Java Program: Successful
***&Java Program:successful**)
this cobol program: successfulBest
that PHP Program: unsuccessful
-----Original Message-----
```

Diff when a word boundary is used

Not highlighted

Captures all word boundaries

**REGULAR EXPRESSION** 60416 matches, 432938 steps (~618ms)

`/ \b`

**TEST STRING** SWITCH TO UNIT TESTS

```
Comment out all Java Program: successful
Java Program: unsuccessful
Java Program: Successful
***&Java Program:successful**)
this cobol program: successfulBest
that PHP Program: unsuccessful
```

All non-word boundary characters.

The screenshot shows a regex testing interface. The 'REGULAR EXPRESSION' field contains `\Bsuccessful\b`. The 'TEST STRING' field contains the following text:

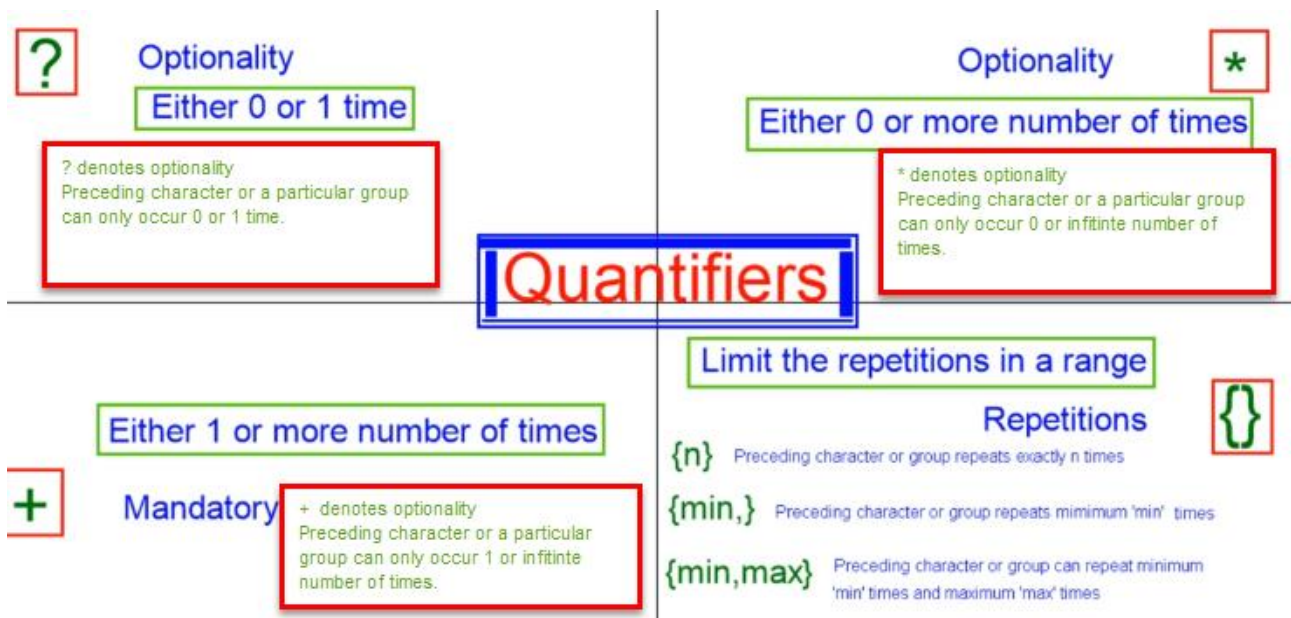
```
Comment out all Java Program: successful
Java Program: unsuccessful
Java Program: Successful
****&JavaJ Program:successful***)
this cobol program: successfulBest
that PHP Program: unsuccessful
```

The result shows '1 match, 65 steps (~0ms)' and highlights the word 'successful' in the test string. A red callout bubble points to the word 'successful' with the text: 'Non words that only have trailing and leading words'.

## 1.5 Quantifiers

In regx there are 4 types of Quantifiers.

A quantifier is used after a character or group and decides how the character or group before the quantifier will occur.



### 1.5.1 ? quantifier

**REGULAR EXPRESSION**

`abc?`

**TEST STRING**

ab  
abc

This means the letter 'c' can occur once or zero times.

So it will match  
abc  
abc  
ab

**REGULAR EXPRESSION**

`colour?`

**TEST STRING**

color  
colour  
behaviour  
harbour  
humor  
humour  
harbor

**REGULAR EXPRESSION**

`harbour?`

**TEST STRING**

color  
colour  
behaviour  
harbour  
humor  
humour  
harbor

2 matches, 14 steps (~0ms)

/ gmi

SWITCH TO UNIT TESTS

REGULAR EXPRESSION: `file[\d]?\\b`

TEST STRING: file, file1, file2, file4, file44, file68, fileab, fileac

REGULAR EXPRESSION: `file[\w]?\\b`

TEST STRING: file, file1, file2, file4, file44, file68, fileab, fileac

REGULAR EXPRESSION: `file[\S]?\\b`

TEST STRING: file, file1, file2, file4, file44, file68, fileab, fileac

This can be achieved using:  
1) Anchors  
2) Quantifiers  
3) WordBoundary

This can be achieved using three different ways.

In this example I want to match the word `file` or `file1` or `file2` or `file3` or `file4`

### 1.5.2 \* Quantifier

REGULAR EXPRESSION: `file\d*\\b`

6 matches, 72 steps (~0ms)

TEST STRING: file, file1, file2, file4, file44, file68, fileab, fileac

In this example I want to match the word `file` and the trailing numbers can be single or multiple digits

REGULAR EXPRESSION: `file[A-z]*\\b`

4 matches, 75 steps (~0ms)

TEST STRING: file, file1, file2, file4, file44, file68, fileab, fileac, fileAbCf

REGULAR EXPRESSION: `file[A-z]*$`

TEST STRING: file, file1, file2, file4, file44, file68, fileab, fileac, fileAbCf

Different ways to achieve this with what ever we have learnt so are:  
1) `/file[A-z]*\\b/gmi`  
2) `/^file[A-z]*$/gmi`

In this example I want to match the word `file` and the trailing characters can be single or multiple letters

### 1.5.3 Matching Example

**Q.1) Create a Regular Expression to match a pattern like below -**

User has to provide a valid Name of company in an on line Form and it can have any alphanumeric value.

Hint:

-----

Name of a company can contain alphanumeric characters

Assumption:

-----

Here, we assume that the names of the company does not contain any other special characters. If your requirement is to match a company name with special characters as well, then the RegeX will change accordingly.

Different waves to achieve it.

The screenshot shows a web-based Regular Expression testing interface. At the top, the 'REGULAR EXPRESSION' field contains the pattern `^\w[\w]*$`. To the right of this field, a status bar indicates '16 matches, 150 steps (~0ms)'. Below the pattern field is a 'TEST STRING' section with a 'SWITCH TO UNIT TESTS' button. The test strings are listed in a text area, with matching strings highlighted in blue. The matching strings are: 'y1', 'a', 'b', 'Google', '21st Century fox', '3M', '888 Enterprises', '1800flowers', '1and1', 'Big 2 Toyota', 'Big 5 Sporting Goods', '21st Century Fox', 'Forever 21', 'four peaks brewery', and 'Brothers Four Car Wash'. Below these, a section titled 'Examples of Non Matching Strings are:' lists strings that do not match the pattern: '1#', '\$4f', and 'k\*b'.

```
REGULAR EXPRESSION: ^\w[\w]*$ 16 matches, 150 steps (~0ms) / gmi
```

TEST STRING SWITCH TO UNIT TESTS ▶

```
s
y1
a
b
Google
21st Century fox
3M
888 Enterprises
1800flowers
1and1
Big 2 Toyota
Big 5 Sporting Goods
21st Century Fox
Forever 21
four peaks brewery
Brothers Four Car Wash
Examples of Non Matching Strings are:
-----
1#
$4f
k*b
```

regular expressions 101

REGULAR EXPRESSION 1 match, 22 steps (~1ms)

`/^[A-z\d][A-z\d\s]*$/gm`

TEST STRING SWITCH TO UNIT TESTS

s  
y1  
a  
b  
google  
21st Century fox  
3M  
888 Enterprises  
1800flowers  
1and1  
Big 2 Toyota  
Big 5 Sporting Goods  
21st Century Fox  
Forever 21  
four peaks brewery  
Brothers Four Car Wash  
=====

1#  
\$4f  
k\*b

REGULAR EXPRESSION

1 match, 97 steps (~1ms)

/ ^[0-9][A-z\d\s]\*\$

/ gmi

TEST STRING

SWITCH TO UNIT TESTS

s  
y1  
a  
b  
Google  
21st Century fox  
3M  
888 Enterprises  
1800flowers  
1and1  
Big 2 Toyota  
Big 5 Sporting Goods  
21st Century Fox  
Forever 21  
four peaks brewery  
Brothers Four Car Wash  
Examples of Non Matching Strings are:  
-----  
1#  
\$4f  
k\*b=

```

<!DOCTYPE html>
<html>
<head>
<title>Testing Regular Expressions in HTML forms</title>
</head>
<body>
<h1>Test your Regular Expressions here</h1>
<fieldset>
<legend>Write your Regular expressions</legend>
<form>
<div>
<label for="display-name"> Enter a company name here:
<span class="warning">*(Enter an alpha numeric character.)</span>
</label>
<input type="text" id="display-name" name="ip-display"
pattern="^[0-9][A-z\d\s]*$"
title="Special Characters Not Allowed"/>
</div>
<div>
<input type="submit" class="submit" value="Submit" />
</div>
</form>
</fieldset>
</body>
</html>

```

Command explanation.



$$^ [A-z \backslash d] [A-z \backslash d \backslash s] * \$$$

$$^ [\backslash w] [\backslash w ] * \$$$

Part 1 will check if the characters are stating with words or numbers and not special characters

Part 2 will check if the string will contain characters, numbers and space " " using the \* quantifier. Which means they can occur 0-infinite

Testing Regular Expressions in HT

File | file:///C:/Users/varun.kumar/Downloads/WarrantySubmit.html?ip-display=1234

## Test your Regular Expressions here

Write your Regular expressions

Enter a company name here: \*(Enter an alpha numeric character.)

Technochron#

Submit



Please match the requested format.  
Special Characters Not Allowed

### 1.5.4 + Quantifier

REGULAR EXPRESSION	REGULAR EXPRESSION	REGULAR EXPRESSION	REGULAR EXPRESSION
/ file[a-zA-Z]	/ file\w	/ file\d	/ file[a-zA-Z]
TEST STRING	TEST STRING	TEST STRING	TEST STRING
file	file	file	file
file1	file1	file1	file1
file2	file2	file2	file2
file3	file3	file3	file3
file4	file4	file4	file4
file44	file44	file44	file44
file68	file68	file68	file68
file86765	file86765	file86765	file86765
file4432	file4432	file4432	file4432
fileab	fileab	fileab	fileab
fileac	fileac	fileac	fileac
fileAB	fileAB	fileAB	fileAB

## 1.5.5 { Quantifiers To Limit Range

The first screenshot shows the regular expression `/1{3}/` applied to a test string containing '1' repeated three times. A green box highlights the match with the text: "Match 1 occurring 3 times".

The second screenshot shows the regular expression `/1{3,5}/` applied to the same test string, matching the '1' repeated three times.

The third screenshot shows the regular expression `/1{3,5}/` applied to a test string containing '1' repeated five times, matching the '1' repeated three times.

The fourth screenshot shows the regular expression `/[oh]{5}/` applied to a test string containing 'o' and 'h' repeated five times. A red callout box explains: "Indicates either o or h can occur 5 times." Another red callout box explains: "Combination of 'o' or 'h' can occur 5 times in any combination." A match detail box shows: "Match 4: group 0: 'hhho' pos: 64-69".

## 1.5.6 {min,} Quantifiers

The screenshot shows the regular expression `/file\w{3,}/` applied to a test string containing various file names. A red callout box explains: "Indicates that after the word 'file' there have to be a minimum of 3 words and maximum can be any number".

1

### 1.5.8 Greedy Quantifiers

REGULAR EXPRESSION 4 matches, 78 steps (~0ms)

`/[oh]{3,5}/gmi`

TEST STRING SWITCH TO UNIT TESTS ▶

```
fileabc
file1
file2
file432
file44rt
file68
fileab
fileacsdffgdfdsd
ohhhhooohhhhhhhhhh
```

Match o or h minimum of 3 times and maximum of 5 times.

There are 4 matches

They are greedy in nature since they will always try to match the longest possible pattern

REGULAR EXPRESSION

`/<.+/`

TEST STRING

```
<>
<!DOCTYPE html>
<html>
<head>
<title> PageTitle </title> //title ends here
</head>
<*h1>Syntax error </h1> I
<style>
<body>
<h1>This is a Heading</h1>
<p>This is a paragraph.</p>
</body>
<?php
echo "My first PHP script!";
?>
</body>
</html>
```

There is a fix for the greedy natures of the quantifiers by making the quantifiers “Lazy” or “Reluctant” quantifiers.

### 1.5.9 Lazy Or Reluctant Quantifiers

The screenshot displays a regex testing interface with two panels. The top panel shows a test for the regex `/<.+>/gm` against an HTML string. It highlights the greedy quantifier `.+` and shows 13 matches. The bottom panel shows the same test but with the reluctant quantifier `.+?`, resulting in 17 matches. Green boxes and red outlines are used to highlight specific parts of the interface and the quantifiers themselves.

**Greedy Quantifiers**

**Reluctant Quantifiers**

**GREEDY QUANTIFIER**

**RELUCTANT QUANTIFIER**

In this example want to match the words only within the double quotes ""

### 1.5.10 Alternative For Lazy Quantifiers

When making a quantifier Lazy this will result in increased processing time and this when used in applications(bulding live applications) will cause lags. This can be over come by using negation characters.

**REGULAR EXPRESSION** 9 matches, 75 steps (~0ms)

REGEX: `/<.*?>/gm`

**TEST STRING** SWITCH TO UNIT TESTS

```

<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>

```

**LAZY QUANTIFIERS**

**REGULAR EXPRESSION** 9 matches, 36 steps (~0ms)

REGEX: `/<[^>]*>/gm`

**TEST STRING** SWITCH TO UNIT TESTS

```

<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>

<p>My first paragraph.</p>

</body>
</html>

```

Lazy Quantifier using negation character and faster processing.

**REGULAR EXPRESSION** 3 matches, 12 steps (~0ms)

REGEX: `/"\w+"/gm`

**TEST STRING** SWITCH TO UNIT TESTS

"Trump" is the "President" of "USA" he is awesome.

**REGULAR EXPRESSION** 3 matches, 26 steps (~0ms)

REGEX: `/"\.+?"/gm`

**TEST STRING** SWITCH TO UNIT TESTS

"Trump" is the "President" of "USA" he is awesome.

**REGULAR EXPRESSION** 3 matches, 12 steps (~0ms)

REGEX: `/"[^"]+"/gm`

**TEST STRING** SWITCH TO UNIT TESTS

"Trump" is the "President" of "USA" he is awesome.



### 1.5.11 Greedy Quantifiers vs Lazy Quantifiers

**REGULAR EXPRESSION** 4 matches, 92 steps (~0ms)

`/s.*o`

**TEST STRING** SWITCH TO UNIT TESTS

```
stackoverflow
abcdee

"Trump" is the "President" of "USA" he is awesome.

=====
Greedy Quantifier - Matches the longest possible string.

Lazy Quantifier - Matches the shortest possible string.
```

**GREEDY, since it is matching from s to last "o"**

**REGULAR EXPRESSION** 1 match, 10 steps (~0ms)

`/abcde?`

**TEST STRING** SWITCH TO UNIT TESTS

```
stackoverflow
abcdee

"Trump"

=====
Greedy Q

Lazy Qua
```

In this scenario if you observe the lazy quantifier will try to match minimum number of times.

Since \* is 0 to infinite, the lazy quantifier is trying to match 'e' 0 number of times.

**EXPLANATION**

- `/abcde?` / gm
  - `abcd` matches the characters `abcd` literally (case sensitive)
  - `e?` matches the character `e` literally (case sensitive)
  - Quantifier** — Matches between zero and unlimited times, as few times as possible, expanding as needed (*lazy*)
  - Global pattern flags**
    - `g` modifier: global. All matches (don't return after first match)
    - `m` modifier: multi line. Causes `^` and `$` to match the begin/end of each line (not only begin/end of string)

**MATCH INFORMATION**

Match 1

Full match 14-18 "abcd"

REGULAR EXPRESSION	REGULAR EXPRESSION
/ abcde.*? I	/ abcde.*? I
TEST STRING	TEST STRING
stackoverflow abcdee	stackoverflow abcdee

## Which is better - Greedy Vs Lazy

$aaab \rightarrow \text{match string}$   
 $a*b \rightarrow \underline{aaa}b$   
 $a*?b \rightarrow a\underline{aab}$

Depends on case to case which is better to use. In this example greedy is better than Lazy, since the match comes in single step

$\underline{ab}$   
 $\underline{aab}$       $\underline{aaab}$

REGULAR EXPRESSION	1 match, 3 steps (~1ms)
/ a*b	/ gm
TEST STRING	SWITCH TO UNIT TESTS
aaab	

REGULAR EXPRESSION	1 match, 6 steps (~0ms)
/ a.*?b	/ gm
TEST STRING	SWITCH TO UNIT TESTS
aaab	

In the pattern match of xml tags we have seen earlier the Lazy quantifiers are much better than greedy quantifiers.

### 1.5.12 Increasing the performance of the RegeX

The normal behavior of a quantifier is always Greedy and in some cases, Greedy quantifiers can lead to performance issues and in some cases, Lazy Quantifier can also lead to performance issues.

#### Note:

If we find that Greedy Quantifiers have performance issue in your search pattern, then try changing it to Lazy Quantifier and see if it improves the performance. On the other hand, if you have a lazy Quantifier which has a performance issue, try to change it into Greedy Quantifier to see if the performance increases. We can also try to use the Negation character to improve the performance.

The key to performance is to always remember -



**\*\* Greedy matches the longest possible string**

**\*\* Lazy matches the smallest possible string**

So, remember this, if we try to make Greedy quantifier as Lazy then the meaning will change like this -

**{min,max}?** - Repeat minimum 'min' times and maximum 'max' times, but as few times as possible(lowest is 'min' times)

**{min,}?** - Repeat minimum 'min' times and maximum any times, but as few times as possible(lowest is 'min' times)

**\*?** - Repeat any number of times, but as few times as possible(lowest is 0 time)

**+?** - Repeat any number of times, but as few times as possible(lowest is 1 time)

**??** - Repeat either 0 time or 1 time, but as few times as possible(lowest is 0 time)

## 1.6 Groups In Regex

To Be Updated

## 1.7 Assertions

To Be Updated

## 1.8 Real Life Scenarios

To Be Updated