10 min read

# Regex Tutorial: Learn With Regular Expression Examples

by **alesanchezr**

Regular Expressions are the best way to identify patterns within strings. They can seem difficult and annoying, but once you know how to use them, they're amazing! In this Regex Tutorial you will learn with Regular Expression Examples

# What is a Regular Expression (Regex)?

Basically, a regular expression is a pattern describing a certain amount of text. For example, you know that emails are always like:

```
username@domain.extension
```

**If we want to describe the pattern of an email, we will say something like this:** Starting with a username (a combination of letters and numbers), followed by an `at` **@** symbol, followed by the domain (another combinations of letters and numbers) followed by the extension (that starts with a dot `.` followed by a combination only letters).

The process of describing the pattern of an email is the same process you will follow when you want to create a regular expression. The only difference will be the syntax.

# Why Use Regex?

All major programming languages use regular expressions (C++, PHP, .NET, Java, JavaScript, Python, Ruby, and many others). As a web developer, you have to always be working with strings to validate the user's inputted data, to validate URL formats, to replace words in paragraphs, etc. These are the main uses for regular expressions:

- **Search** for particular items within a large body of text. For example, you may wish to identify all email addresses in some content using a text editor.
- **Replace** particular items. For example, you may wish to clean up some poorly formatted HTML by replacing all uppercase tags with lowercase equivalents in a text editor.
- **Validate** input. For example, you may want to check that a password meets certain criteria such as: a mix of uppercase and lowercase, digits, punctuation, etc., in a program that you are writing.
- **Coordinate** actions. For example, you may wish to process certain files in a directory, but only if they meet particular conditions. In work you are doing so on the command line.
- **Reformat** text. For example, you may export data from one program as a text file, then modify its layout so that you can import it into another program using a text editor.
- and more…

# Building and Testing Regular Expressions

Never start creating a Regex without having a live testing tool – it can get very complicated very easily. The best way is to use the "divide and conquer" strategy (again) – split your Regex into several smaller Regex's, and then combine them all.

# The Patterns Syntax

This is a regular expression that checks for an email pattern:

```jsx
/[\w._%+-]+@[\w.-]+\.[a-zA-Z]{2,4}/
```

But, don't worry…you don't have to understand it right now. The good news is that a complex regular expression is just the combinations of several very simple regular expressions. "Divide and conquer!"

So…let's start with the basic regular expressions using the most basic operators:

## Simple Characters

**A simple character is…**

Any succession of characters is a simple regular expression. If we use the word "email" as a regular expression, the system will look for any repetitions of the word "email" inside of the given text.

**Use the container on the right to play with other simple successions of characters.**

Click to open demo in a new window

## The `.` Character

The `.` character represents…

Any character or symbol available. If you say `ab.ve` you are saying anything that starts with `ab` and ends with `ve`

You can use the `.` as many times as you want; the regular expression will replace the `.` with any character as many times as the `.` appears.

**Use the container on the right to play with other simple successions of characters.**

Click to open demo in a new window

## The Range `[ ]` Character

**The `[ ]` characters represent…**

A group of possible characters. Sometimes, we would like to be a bit more specific…this is where **ranges** come in useful. We specify a range of characters by enclosing them within square brackets ( [ ] ).

You can also use the `[ ]` to range numbers or letters with a dash in between. The dash represents a range of numbers or characters. For example:

- `[0-9]` represents any number between 0 and 9.
- `[a-z]` represents any letter in lowercase
- `[A-Z]` represent any letter in uppercase

You can also combine ranges of characters like this:

- Any letter in uppercase or lowercase: `[a-zA-Z]`
- Numbers from 1 to 5 and also the 9: `[1-59]`
- Numbers from 1 to 5, letters from a to f and also the capital X: `[1-5a-fX]`

Click to open demo in a new window

Click to open demo in a new window

## The `^` (caret) Character: Negation or Beginning of a Term

**If we place `^` at the beginning of a [range]:**

We are negating the range. For example:

- All terms that start with `li` and end with `e` but have no `i` or `v` on the inside: `li[^v]e`

**If we place `^` at the beginning of a regular expression:**

- We are saying that we want to only test the Regex from the beginning of the string (no substrings – smaller parts of the string – will be tested):
- A string starting with http: `^http`

Click to open demo in a new window

## Shortcuts for Digits `\d` and Words `\w`

If you prefer, you can use these shortcuts in your regular expressions:

| Operator | Descriptions |
| --- | --- |
| \w | Matches any word character (equal to `[a-zA-Z0-9_]`) |
| \W | Matches anything other than a letter, digit or underscore |
| \d | Matches any decimal digit. Equivalent to `[0-9]` |
| \D | Matches anything other than a decimal digit |

Click to open demo in a new window

## Grouping or Enclosing Regular Expressions with `()`

We always talk about "divide and conquer," right? Well, your best friend for that will be the parenthesis operator `( )`. We are now able to group any pattern just like we do in math.

Now that we can group, we can multiply (repeat) our patterns, negate our patterns, etc.

For example, this Regex accepts one or many repetitions of the `ab` string followed by a `c` letter at the end: `(ab)*c`

Click to open demo in a new window

## Using Quantifier in Regular Expressions

Sometimes, you don’t want to specify the number of characters that a Regex can have. For example, a domain name can have between 1 to maybe 100 characters…who knows?

Quantifier allow us to increase the number of times a character may occur in our regular expression. Here is the basic set of multipliers:

- * – character occurs zero or more times.

- + – character occurs one or more times.
- ? – character occurs zero or one times.
- {5} – character occurs five times.
- {3,7} – character occurs between 3 and 7 times.
- {2,} – character occurs at least 2 times.

## Using the * + ? Quantifier

We can place the quantifier after the character patterns that we want to repeat. Here are some cases and examples:

| Operator | Description |
| --- | --- |
| + | One or many E.g.:<br>Terms with the letter `o` at least one time; `o+` |
| * | Zero or many<br>E.g.: Terms starting with the letter "a" (lowercase) followed by **zero or many** characters of any type but the white space: `a[^ ]*` |
| ? | Zero or one<br>E.g.: Finding the November string with or without the shortcut: `[nN]ov(ember)?` |

Click to open demo in a new window

> ☝*Here are two amazing tools to build, & test Regular Expressions.https://regex101.com/ and http://regexr.com/*

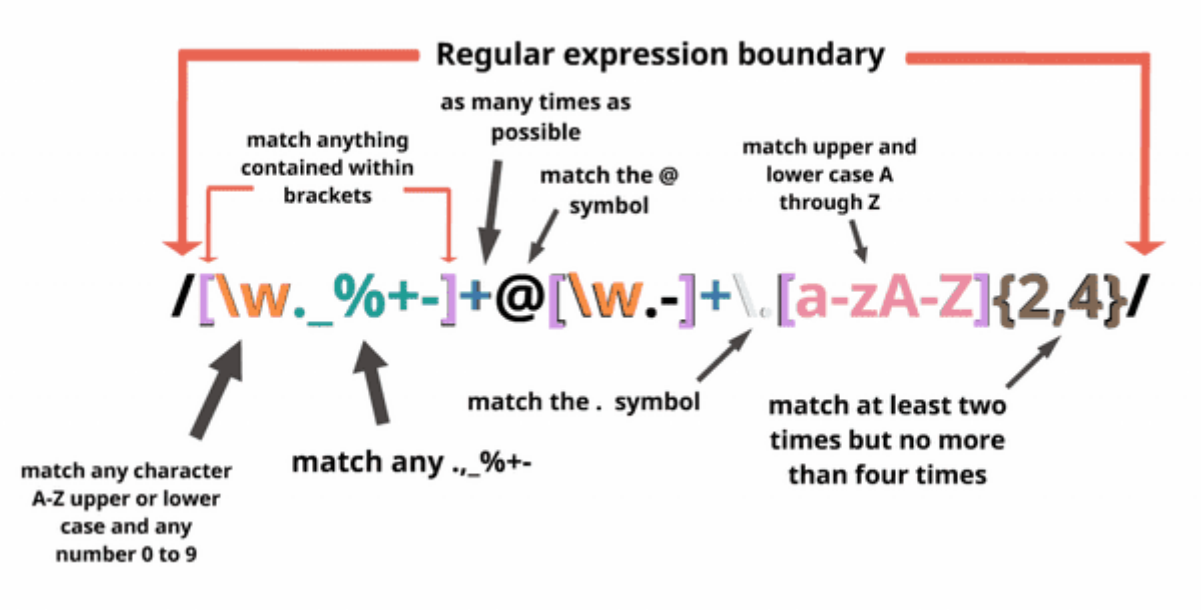
> ☝*Here is an interactive tutorial to learn regular expressions:https://regexone.com/*

## Some very useful Regular Expressions

Lets face it: regular expressions are something you will use every once in a while (unless you specialize in a very particular area of the web development world). The syntax is easy to forget, and you probably are going to find your Regex's from the internet a lot of the time. The important thing here is that you understand them and that you are able to play with them whenever you need to.

Here are some pre-made Regex's:

### Regular Expression for: Validating an Email




We begin by telling the parser to find the beginning of the string (^).

Inside the first group, we match one or more lowercase letters, numbers, underscores, dots, or hyphens.

We have escaped the dot because a non-escaped dot means any character.

Directly after that, there must be an at @ sign.

Next is the domain name, which must be: one or more lowercase letters, numbers, underscores, dots, or hyphens. Then another (escaped) dot, with the extension being two to six letters or dots. I have 2 to 6 because of the country specific TLD's (.ny.us or .co.uk).

Finally, we want the end of the string ($).

Click to open demo in a new window

### Regular Expression for: Matching a URL

This Regex is almost like taking the ending part of the above Regex, slapping it between "http://" and some file structure at the end. It sounds a lot simpler than it really is. To start off, we must search for the beginning of the line with the caret.

The first capturing group is all optional. It allows the URL to begin with "http://", "https://", or neither of them. We have a question mark after the s to allow URL's that have http or https. In order to make this entire group optional, we just added a question mark to the end of it.

Next is the domain name: one or more numbers, letters, dots, or hyphens followed by another dot then two to six letters or dots. The following section is the optional files and directories. Inside the group, we want to match any number of forward slashes, letters, numbers, underscores, spaces, dots, or hyphens. Then we shall say that this group can be matched as many times as we want. This allows multiple directories to be matched along with a file at the end. We have used the star instead of the question mark because the star says zero **or more**, not zero **or one**. If a question mark was to be used there, only one file/directory would be able to be matched.

Next, a trailing slash is matched, but it is optional.

Finally, we end with the end of the line.

Click to open demo in a new window