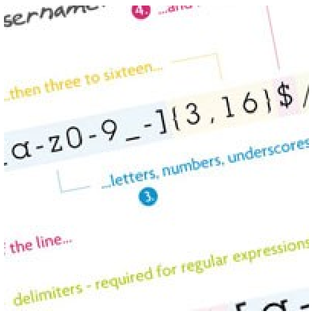


8 Regular Expressions You Should Know

Aug 10th in Other by Vasili

Regular expressions are a language of their own. When you learn a new programming language, they're this little sub-language that makes no sense at first glance. Many times you have to read another tutorial, article, or book just to understand the "simple" pattern described. Today, we'll review eight regular expressions that you should know for your next coding project.

Author: Vasili

This is a NETTUTS contributor who has published 2 tutorial(s) so far here. Their bio is coming soon!

Background Info on Regular Expressions

This is what Wikipedia has to say about them:

> In computing, regular expressions provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters. Regular expressions (abbreviated as regex or regexp, with plural forms regexes, regexps, or regexen) are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

Now, that doesn't really tell me much about the actual patterns. The regexes I'll be going over today contains characters such as \w, \s, \1, and many others that represent something totally different from what they look like.

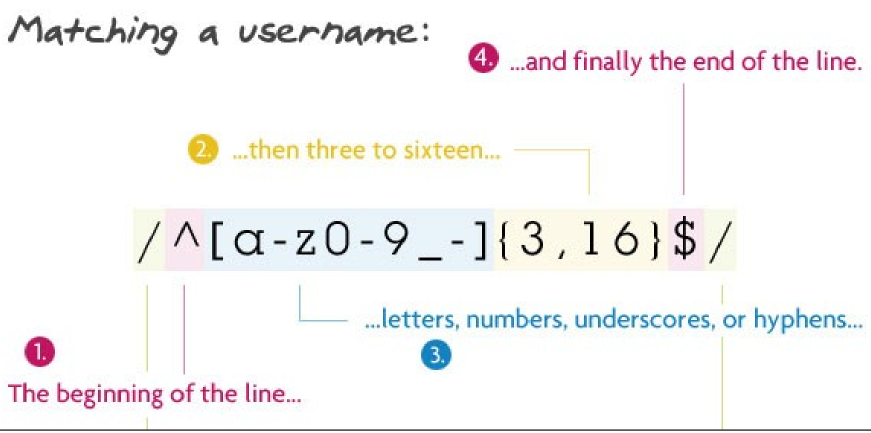
If you'd like to learn a little about regular expressions before you continue reading this article, I'd suggest watching the Regular Expressions for Dummies screencast series.

The eight regular expressions we'll be going over today will allow you to match a(n): username, password, email, hex value (like #fff or #000), slug, URL, IP address, and an HTML tag. As the list goes down, the regular expressions get more and more confusing. The pictures for each regex in the beginning are easy to follow, but the last four are more easily understood by reading the explanation.

The key thing to remember about regular expressions is that they are almost read forwards and backwards at the same time. This sentence will make more sense when we talk about matching HTML tags.

Note: The delimiters used in the regular expressions are forward slashes, "/". Each pattern begins and ends with a delimiter. If a forward slash appears in a regex, we must escape it with a backslash: "\/".

Matching a Username


Matching a username:
4. ...and finally the end of the line.
2. ...then three to sixteen...
/^[a-z0-9_-]{3,16}$/
...letters, numbers, underscores, or hyphens...
3.
1.
The beginning of the line...

Pattern:

```
/^[a-z0-9_-]{3,16}$/
```

Description:

We begin by telling the parser to find the beginning of the string (^), followed by any lowercase letter (a-z), number (0-9), an underscore, or a hyphen. Next, {3,16} makes sure that are at least 3 of those characters, but no more than 16. Finally, we want the end of the string ($).

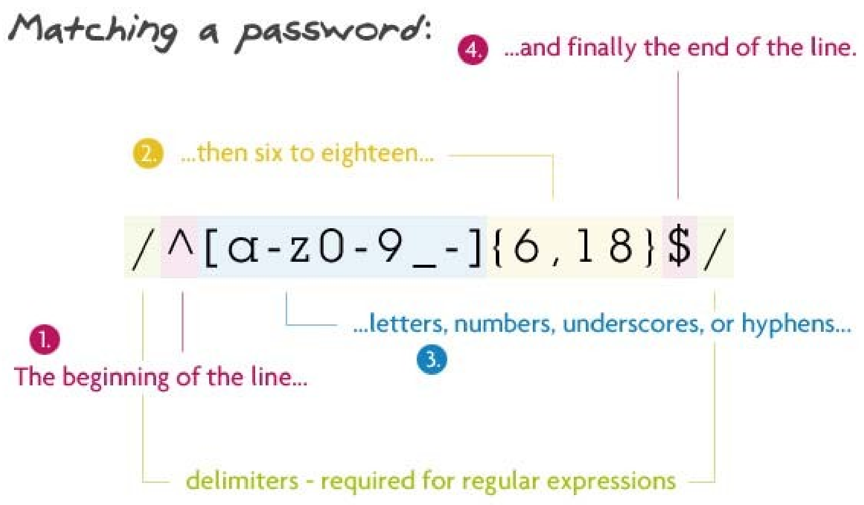
String that matches:

my-us3r_n4m3

String that doesn't match:

th1s1s-wayt00_l0ngt0beausername (too long)

Matching a Password


Matching a password:
4. ...and finally the end of the line.
2. ...then six to eighteen...
/^[a-z0-9_-]{6,18}$/
...letters, numbers, underscores, or hyphens...
3.
1. The beginning of the line...
delimiters - required for regular expressions

Pattern:

```
/^[a-z0-9_-]{6,18}$/
```

Description:

Matching a password is very similar to matching a username. The only difference is that instead of 3 to 16 letters, numbers, underscores, or hyphens, we want 6 to 18 of them ({6,18}).

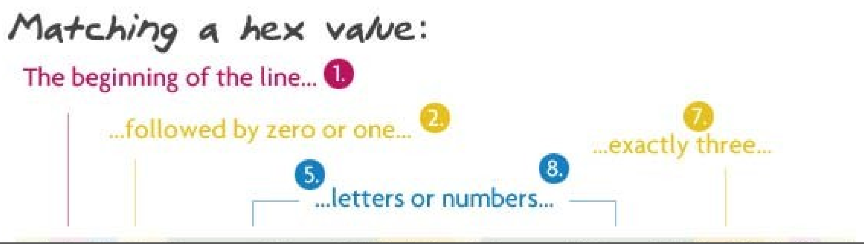
String that matches:

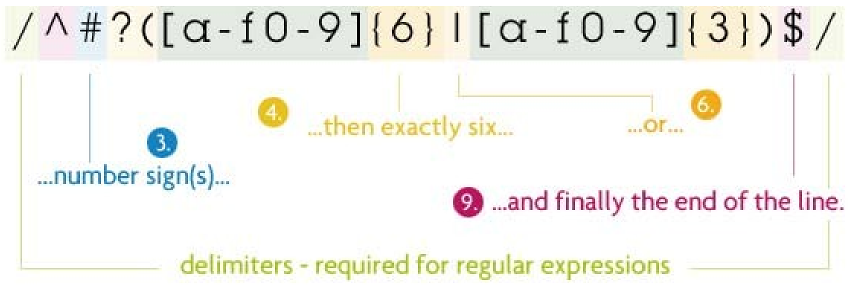
myp4ssw0rd

String that doesn't match:

mypa$$w0rd (contains a dollar sign)

Matching a Hex Value


Matching a hex value:
The beginning of the line... 1.
...followed by zero or one... 2.
7. ...exactly three...
5. ...letters or numbers... 8.


/^#?([a-f0-9]{6}|[a-f0-9]{3})$/
3. ...number sign(s)...
4. ...then exactly six...
...or... 6.
9. ...and finally the end of the line.
delimiters - required for regular expressions

Pattern:

```
/^#?([a-f0-9]{6}|[a-f0-9]{3})$/
```

Description:

We begin by telling the parser to find the beginning of the string (^). Next, a number sign is optional because it is followed a question mark. The question mark tells the parser that the preceding character — in this case a number sign — is optional, but to be "greedy" and capture it if it's there. Next, inside the first group (first group of parentheses), we can have two different situations. The first is any lowercase letter between a and f or a number six times. The vertical bar tells us that we can also have three lowercase letters between a and f or numbers instead. Finally, we want the end of the string ($).

The reason that I put the six character before is that parser will capture a hex value like #ffffff. If I had reversed it so that the three characters came first, the parser would only pick up #fff and not the other three f's.

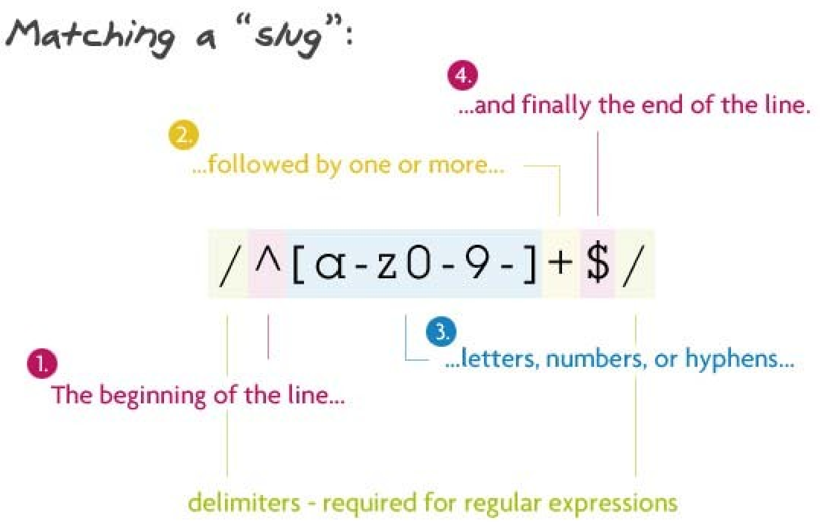
String that matches:

#a3c113

String that doesn't match:

#4d82h4 (contains the letter h)

Matching a Slug


Matching a "slug":
4. ...and finally the end of the line.
2. ...followed by one or more...
/^[a-z0-9-]+$/
1. The beginning of the line...
3. ...letters, numbers, or hyphens...
delimiters - required for regular expressions

Pattern:

```
/^[a-z0-9-]+$/
```

Description:

You will be using this regex if you ever have to work with mod_rewrite and pretty URL's. We begin by telling the parser to find the beginning of the string (^), followed by one or more (the plus sign) letters, numbers, or hyphens. Finally, we want the end of the string ($).

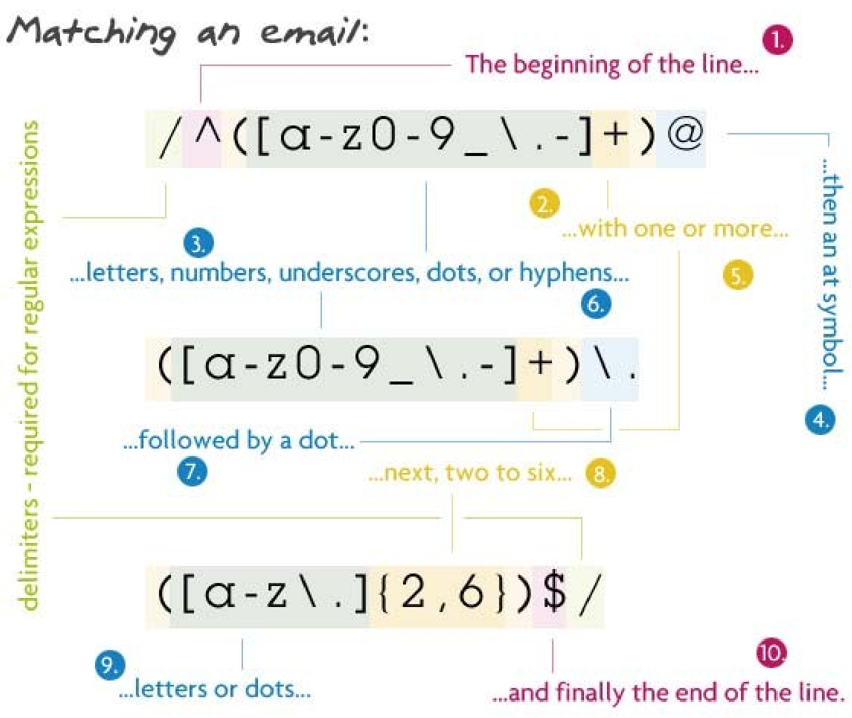
String that matches:

my-title-here

String that doesn't match:

my_title_here (contains underscores)

Matching an Email



Pattern:

```
/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})$/
```

Description:

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. I 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 ($).

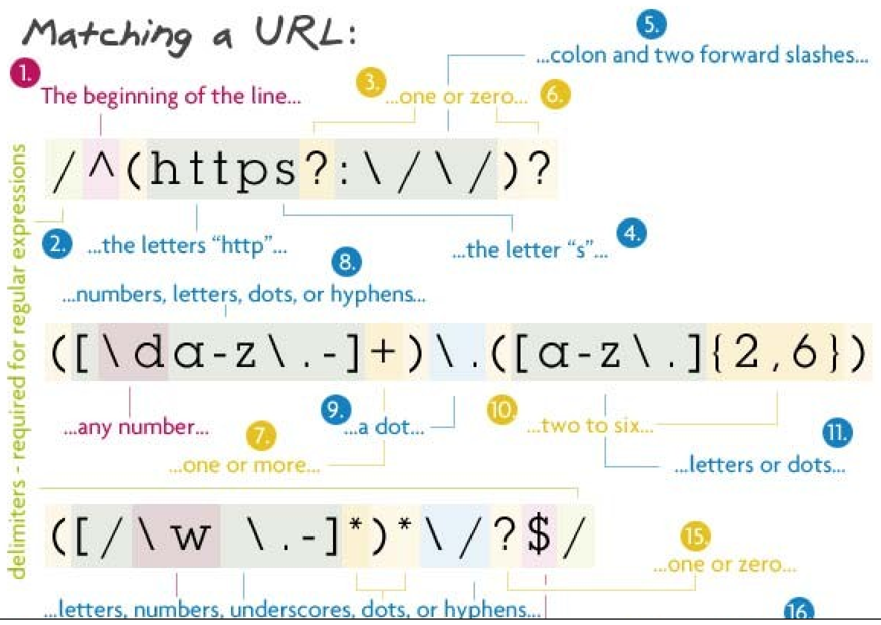
String that matches:

john@doe.com

String that doesn't match:

john@doe.something (TLD is too long)

Matching a URL



13.
12.
...zero or more...
14.
...a forward slash...
17.
...any letter, number, or hyphen...
...and finally the end of the line.

Pattern:

```
/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]*)*\/?$/
```

Description:

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 search for the beginning of the line with the caret.

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

Next is the domain name: one or more numbers, letters, dots, or hypens 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 say that this group can be matched as many times as we want. Pretty much this allows multiple directories to be matched along with a file at the end. I 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.

Then a trailing slash is matched, but it can be optional. Finally we end with the end of the line.

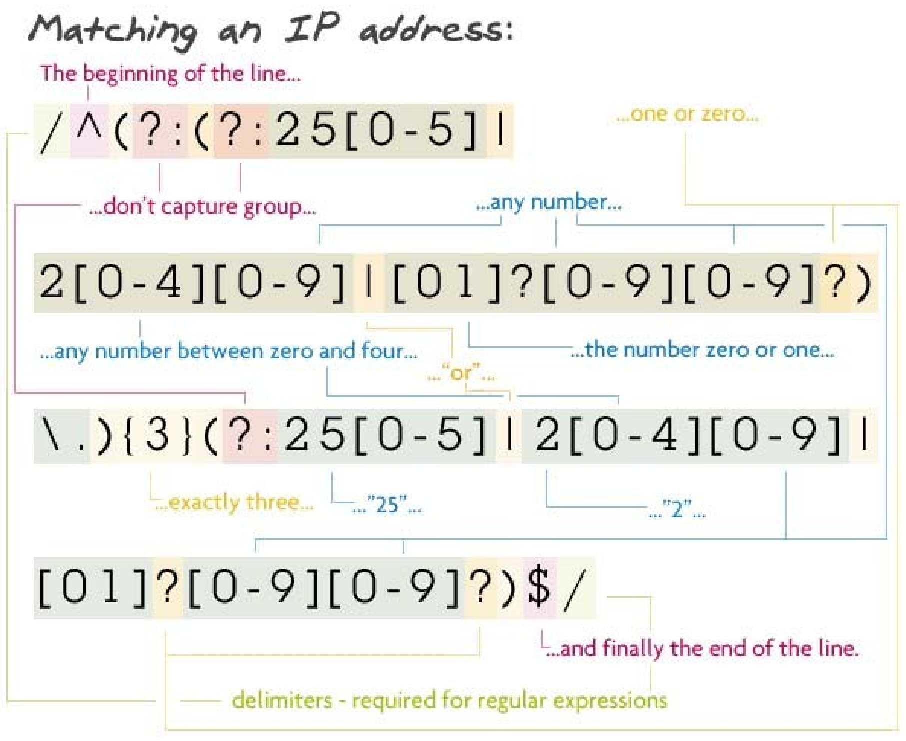
String that matches:

http://net.tutsplus.com/about

String that doesn't match:

http://google.com/some/file!.html (contains an exclamation point)

Matching an IP Address



Matching an IP address:
The beginning of the line...
/^(?:(?:25[0-5]|
...one or zero...
...don't capture group...
...any number...
2[0-4][0-9]|[01]?[0-9][0-9]?)
...any number between zero and four...
..."or"...
...the number zero or one...
\.){3}(?:25[0-5]|2[0-4][0-9]|
...exactly three...
..."25"...
..."2"...
[01]?[0-9][0-9]?)$/
...and finally the end of the line.
delimiters - required for regular expressions

Pattern:

```
/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/
```

Description:

Now, I'm not going to lie, I didn't write this regex; I got it from here. Now, that doesn't mean that I can't rip it apart character for character.

The first capture group really isn't a captured group because

```
?:
```

was placed inside which tells the parser to not capture this group (more on this in the last regex). We also want this non-captured group to be repeated three times — the {3} at the end of the group. This group contains another group, a subgroup, and a literal dot. The parser looks for a match in the subgroup then a dot to move on.

The subgroup is also another non-capture group. It's just a bunch of character sets (things inside brackets): the string "25" followed by a number between 0 and 5; or the string "2" and a number between 0 and 4 and any number; or an optional zero or one followed by two numbers, with the second being optional.

After we match three of those, it's onto the next non-capturing group. This one wants: the string "25" followed by a number between 0 and 5; or the string "2" with a number between 0 and 4 and another number at the end; or an optional zero or one followed by two numbers, with the second being optional.

We end this confusing regex with the end of the string.

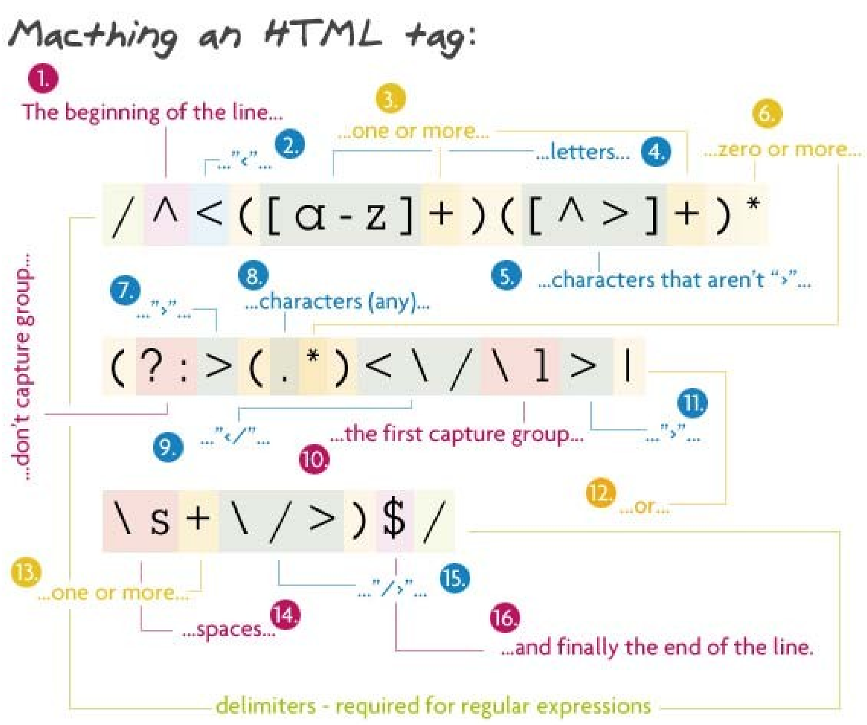
String that matches:

73.60.124.136 (no, that is not my IP address :P)

String that doesn't match:

256.60.124.136 (the first group must be "25" and a number between zero and **five**)

Matching an HTML Tag


Macthing an HTML tag:
1. The beginning of the line...
2. ..."<"...
3. ...one or more...
4. ...letters...
5. ...characters that aren't ">"...
6. ...zero or more...
/ ^ < ([a - z] +) ([^ >] +) *
7. ...">"...
8. ...characters (any)...
(? : > (. *) < \ / \ 1 > |
9. ..."</"...
10. ...the first capture group...
11. ...">"...
12. ...or...
\ s + \ / >) $ /
13. ...one or more...
14. ...spaces...
15. ..."/>"...
16. ...and finally the end of the line.
...don't capture group...
delimiters - required for regular expressions

Pattern:

```
/^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$/
```

Description:

One of the more useful regexes on the list. It matches any HTML tag with the content inside. As usually, we begin with the start of the line.

First comes the tag's name. It must be one or more letters long. This is the first capture group, it comes in handy when we have to grab the closing tag. The next thing are the tag's attributes. This is any character but a greater than sign (>). Since this is optional, but I want to match more than one character, the star is used. The plus sign makes up the attribute and value, and the star says as many attributes as you want.

Next comes the third non-capture group. Inside, it will contain either a greater than sign, some content, and a closing tag; or some spaces, a forward slash, and a greater than sign. The first option looks for a greater than sign followed by any number of characters, and the closing tag. \1 is used which represents the content that was captured in the first capturing group. In this case it was the tag's name. Now, if that couldn't be matched we want to look for a self closing tag (like an img, br, or hr tag). This needs to have one or more spaces followed by "/>".

The regex is ended with the end of the line.

String that matches:

<a href="http://net.tutsplus.com/">Nettuts+</a>

String that doesn't match:

<img src="img.jpg" alt="My image>" /> (attributes can't contain greater than signs)

Conclusion

I hope that you have grasped the ideas behind regular expressions a little bit better. Hopefully you'll be using these regexes in future projects! Many times you won't need to decipher a regex character by character, but sometimes if you do this it helps you learn. Just remember, don't be afraid of regular expressions, they might not seem it, but they make your life a lot easier. Just try and pull out a tag's name from a string without regular expressions! ;)

- Follow us on Twitter, or subscribe to the NETTUTS RSS Feed for more daily web development tuts and articles.