You Don’t Know Anything About Regular Expressions: A Complete Guide

Regular expressions can be scary…really scary. Fortunately, once you memorize what each symbol represents, the fear quickly subsides. If you fit the title of this article, there’s much to learn! Let’s get started.

Section 1: Learning the Basics

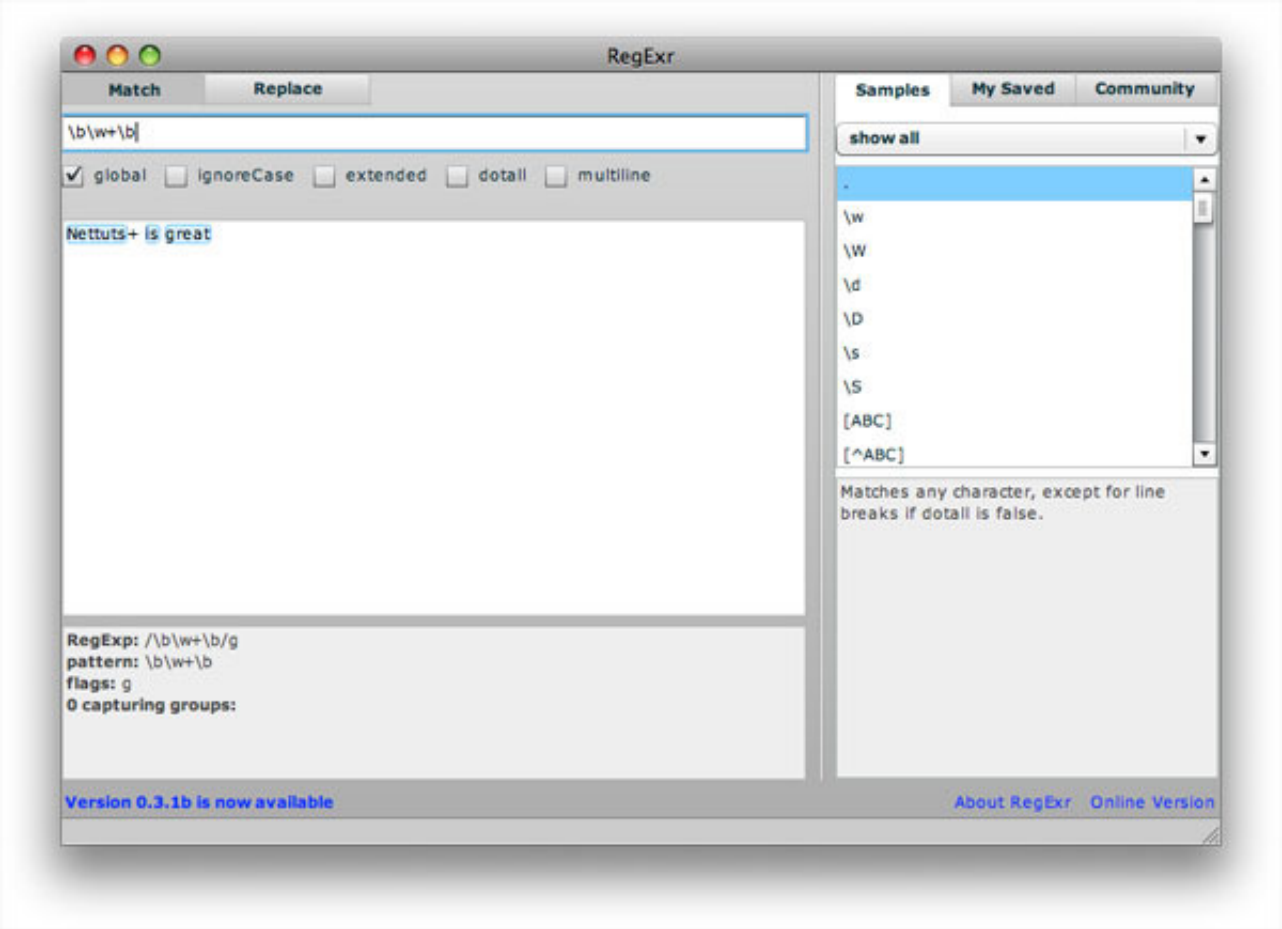
The key to learning how to effectively use regular expressions is to just take a day and memorize all of the symbols. This is the best advice I can possibly offer. Sit down, create some flash cards, and just memorize them! Here are the most common:

- **.** – Matches any character, except for line breaks if dotall is false.
- * – Matches 0 or more of the preceding character.
- + – Matches 1 or more of the preceding character.
- **?** – Preceding character is optional. Matches 0 or 1 occurrence.
- **\d** – Matches any single digit
- **\w** – Matches any word character (alphanumeric & underscore).
- **[XYZ]** – Matches any single character from the character class.
- **[XYZ]+** – Matches one or more of any of the characters in the set.
- **$** – Matches the end of the string.
- ^ – Matches the beginning of a string.
- **[^a-z]** – When inside of a character class, the ^ means NOT; in this case, match anything that is NOT a lowercase letter.

Yep – it’s not fun, but just memorize them. You’ll be thankful if you do!

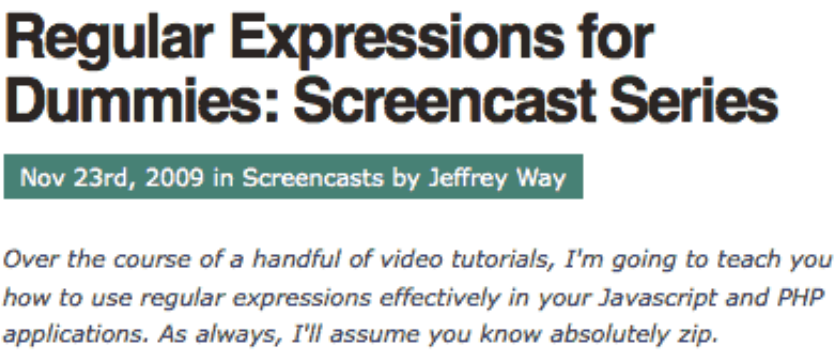
Tools

You can be certain that you’ll want to rip your hair out at one point or another when an expression doesn’t work, no matter how much it should – or you think it should! Downloading the RegExr Desktop app is essential, and is really quite fun to fool around with. In addition to real-time checking, it also offers a sidebar which details the definition and usage of every symbol. Download it!.



Section 2: Regular Expressions for Dummies: Screencast Series

The next step is to learn how to actually use these symbols! If video is your preference, you're in luck! Watch the five lesson video series, "Regular Expressions for Dummies."



Section 3: Regular Expressions and JavaScript

In this final section, we'll review a handful of the most important JavaScript methods for working with regular expressions.

1. Test()

This one accepts a single string parameter and returns a boolean indicating whether or not a match has been found. If you don't necessarily need to perform an operation with the a specific matched result – for instance, when validating a username – "test" will do the job just fine.

Example

```
var username = 'JohnSmith';
alert(/[A-Za-z_-]+/.test(username)); // returns true
```

Above, we begin by declaring a regular expression which only allows upper and lower case letters, an underscore, and a dash. We wrap these accepted characters within brackets, which designates a **character class**. The "+" symbol, which proceeds it, signifies that we're looking for one or more of any of the preceding characters. We then test that pattern against our variable, "JohnSmith." Because there was a match, the browser will display an alert box with the value, "true."

2. Split()

You're most likely already familiar with the split method. It accepts a single regular expression which represents where the "split" should occur. *Please note that we can also use a string if we'd prefer*.

```
var str = 'this is my string';
alert(str.split(/\s/)); // alerts "this, is, my, string"
```

By passing "\s" – representing a single space – we've now split our string into an array. If you need to access one particular value, just append the desired index.

```
var str = 'this is my this string';
alert(str.split(/\s/)[3]); // alerts "string"
```

3. Replace()

As you might expect, the "replace" method allows you to replace a certain block of text, represented by a string or regular expression, with a different string.

Example

If we wanted to change the string "Hello, World" to "Hello, Universe," we could do the following:

```
var someString = 'Hello, World';
someString = someString.replace(/World/, 'Universe');
alert(someString); // alerts "Hello, Universe"
```

It should be noted that, for this simple example, we could have simply used .replace('World', 'Universe'). Also, using the replace method does not automatically overwrite the value the variable, we must reassign the returned value back to the variable, someString.

Example 2

For another example, let's imagine that we wish to perform some elementary security precautions when a user signs up for our fictional site. Perhaps we want to take their username and remove any symbols, quotation marks, semi-colons, etc. Performing such a task is trivial with JavaScript and regular expressions.

```
var username = 'J;ohnSmith;@%';
username = username.replace(/[^A-Za-z\d_-]+/, '');
alert(username); // JohnSmith;@%
```

Given the produced alert value, one might assume that there was an error in our code (which we'll review shortly). However, this is not the case. If you'll notice, the semi-colon immediately after the "J" was removed as expected. To tell the engine to continue searching the string for more matches, we add a "g" directly after our closing forward-slash; this modifier, or **flag**, stands for "global." Our revised code should now look like so:

```
var username = 'J;ohnSmith;@%';
username = username.replace(/[^A-Za-z\d_-]+/g, '');
alert(username); // alerts JohnSmith
```

Now, the regular expression searches the ENTIRE string and replaces all necessary characters. To review the actual expression – **.replace(/[^A-Za-z\d_-]+/g, '');** – it's important to notice the carot symbol inside of the brackets. When placed within a character class, this means "find anything that IS NOT…" Now, if we re-read, it says, find anything that is NOT a letter, number (represented by \d), an underscore, or a dash; if you find a match, replace it with nothing, or, in effect, delete the character entirely.

4. Match()

Unlike the "test" method, "match()" will return an array containing each match found.

Example

```
var name = 'JeffreyWay';
alert(name.match(/e/)); // alerts "e"
```

The code above will alert a single "e." However, notice that there are actually two e's in the string "JeffreyWay." We, once again, must use the "g" modifier to declare a "**g**lobal search.

```
var name = 'JeffreyWay';
alert(name.match(/e/g)); // alerts "e,e"
```

If we then want to alert one of those specific values with the array, we can reference the desired index after the parentheses.

```
var name = 'JeffreyWay';
alert(name.match(/e/g)[1]); // alerts "e"
```

Example 2

Let's review another example to ensure that we understand it correctly.

```
var string = 'This is just a string with some 12345 and some !@#$ mixed in.';
alert(string.match(/[a-z]+/gi)); // alerts
"This,is,just,a,string,with,some,and,some,mixed,in"
```

Within the regular expression, we created a pattern which matches one or more upper or lowercase letters – thanks to the "i" modifier. We also are appending the "g" to declare a global search. The code above will alert "This,is,just,a,string,with,some,and,some,mixed,in." If we then wanted to trap one of these values within the array inside of a variable, we just reference the correct index.

```
var string = 'This is just a string with some 12345 and some !@#$ mixed in.';
var matches = string.match(/[a-z]+/gi);
alert(matches[2]); // alerts "just"
```

Splitting an Email Address

Just for practice, let's try to split an email address – nettuts@tutsplus.com – into its respective username and domain name: "nettuts," and "tutsplus."

```
var email = 'nettuts@tutsplus.com';
alert(email.replace(/([a-z\d_-]+)@([a-z\d_-]+)\.[a-z]{2,4}/ig, '$1, $2')); // alerts
"nettuts, tutsplus"
```

If you're brand new to regular expressions, the code above might look a bit daunting. Don't worry, it did for all of us when we first started. Once you break it down into subsets though, it's really quite simple.

Let's take it piece by piece.

```
.replace(/([a-z\d_-]+)
```

Starting from the middle, we search for any letter, number, underscore, or dash, and match one ore more of them (+). We'd like to access the value of whatever is matched here, so we wrap it within parentheses. That way, we can reference this matched set later!

```
@([a-z\d_-]+)
```

Immediately following the preceding match, find the @ symbol, and then another set of one or more letters, numbers, underscore, and dashes. Once again, we wrap that set within parentheses in order to access it later.

```
\.[a-z]{2,4}/ig,
```

Continuing on, we find a single period (we must escape it with "\" due to the fact that, in regular expressions, it matches any character (sometimes excluding a line break). The last part is to find the ".com." We know that the majority, if not all, domains will have a suffix range of two – four characters (com, edu, net, name, etc.). If we're aware of that specific range, we can forego using a more generic symbol like * or +, and instead wrap the two numbers within curly braces, representing the minimum and maximum, respectively.

```
'$1, $2')
```

This last part represents the second parameter of the replace method, or what we'd like to replace the matched sets with. Here, we're using $1 and $2 to refer to what was stored within the first and second sets of parentheses, respectively. In this particular instances, $1 refers to "nettuts," and $2 refers to "tutsplus."

Creating our Own Location Object

For our final project, we'll replicate the location object. For those unfamiliar, the location object provides you with information about the current page: the href, host, port, protocol, etc. Please note that this is purely for practice's sake. In a real world site, just use the preexisting location object!

We first begin by creating our location function, which accepts a single parameter representing the url that we wish to "decode;" we'll call it "loc."

```
function loc(url) { }
```

Now, we can call it like so, and pass in a gibberish url :

```
var l = loc('http://www.somesite.com?
somekey=somevalue&anotherkey=anothervalue#theHashGoesHere');
```

Next, we need to return an object which contains a handful of methods.

```
function loc(url) {
return {

}
}
```

Search

Though we won't create all of them, we'll mimic a handful or so. The first one will be "search." Using regular expressions, we'll need to search the url and return everything within the querystring.

```
return {
search : function() {
return url.match(/\?(.+)/i)[1];
// returns "somekey=somevalue&anotherkey=anothervalue#theHashGoesHere"
}
}
```

Above, we take the passed in url, and try to match our regular expressions against it. This expression searches through the string for the question mark, representing the beginning of our querystring. At this point, we need to trap the remaining characters, which is why the *(.+)* is wrapped within parentheses. Finally, we need to return only that block of characters, so we use [1] to target it.

Hash

Now we'll create another method which returns the hash of the url, or anything after the pound sign.

```
hash : function() {
return url.match(/#(.+)/i)[1]; // returns "theHashGoesHere"
},
```

This time, we search for the pound sign, and, once again, trap the following characters within parentheses so that we can refer to only that specific subset – with [1].

Protocol

The protocol method should return, as you would guess, the protocol used by the page – which is

generally "http" or "https."

```
protocol : function() {
return url.match(/(ht|f)tps?:/i)[0]; // returns 'http:'
},
```

This one is slightly more tricky, only because there are a few choices to compensate for: http, https, and ftp. Though we could do something like – *(http|https|ftp)* – it would be cleaner to do: *(ht|f)tps?* This designates that we should first find either an "ht" or the "f" character. Next, we match the "tp" characters. The final "s" should be optional, so we append a question mark, which signifies that there may be zero or one instance of the preceding character. Much nicer.

Href

For the sake of brevity, this will be our last one. It will simply return the url of the page.

```
href : function() {
return url.match(/(.+\.[a-z]{2,4})/ig); // returns "http://www.somesite.com"
}
```

Here we're matching all characters up to the point where we find a period followed by two-four characters (representing com, au, edu, name, etc.). It's important to realize that we can make these expressions as complicated or as simple as we'd like. It all depends on how strict we must be.

Our Final Simple Function:

```
function loc(url) {
return {
search : function() {
return url.match(/\?(.+)/i)[1];
},

hash : function() {
return url.match(/#(.+)/i)[1];
},

protocol : function() {
return url.match(/(ht|f)tps?:/)[0];
},

href : function() {
return url.match(/(.+\.[a-z]{2,4})/ig);
}
}
}
```

With that function created, we can easily alert each subsection by doing:

```
var l = loc('http://www.net.tutsplus.edu?key=value#hash');

alert(l.href()); // http://www.net.tutsplus.com
alert(l.protocol()); // http:

...etc.
```

Conclusion

Thanks for reading! I'm Jeffrey Way...signing off.

Original URL:

http://net.tutsplus.com/tutorials/javascript-ajax/you-dont-know-anything-about-regular-expressions/