Regex for people who should know Regex, but do not. PART 2

written by dbr/Ben
http://neverfear.org

Table of Contents

Part 2

This is the second part of the catchily named Regex-for-people-who-should-know-regex-but-do-not guide. If you are unfamiliar with the basics of regular expression syntax, then I recommend you first read part 1, or any of the many regular expressions primers available online.

In this part of the guide, I am first going to cover practical examples of regular expressions, such as how to extract a segment of text from a string, and how to validate user-input. Finally, I am going to cover how to use regex within various languages; how to do regex matching, extraction and substitution within Perl, Ruby, Python, PHP and Javascript.

These examples are going to mainly focus on processing HTML, but remember, HTML is just specifically formatted text. As I keep saying, regular expressions are very flexible - what you learn in this guide can be applied to a myriad of things - from editing files to data-mining, input validation to finding files.
If you are dealing with text (be it code, HTML, file names), chances are regex can help you in some way.

If you don't understand a bit of the article, don't worry - just read past and come back to it later. Learning regular-expressions is just like learning a new language, you start of learning some basics, then learn some more advanced bits. As you learn more advanced bits, the simple bits make more sense. As with all languages, the more you use them, the more familiar you become - think up some projects that could use regular expressions, and get coding.

Practical examples

To actually see why regular expressions are so useful, you really have to have a reason to use them. Since regexs are so flexible, there are many different ways they can be used. I'm going to cover what I use them for most: validating input, cleaning up input, and extracting segments from a string.

Validating form input

This can be a rather complicated matter - "k" in the comments on the previous article corrected pointed out that there was far more to validating an email than just

```
    \w+@\w+\.\w+
```

Look at the full regular expression to validate an email: http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html - and even that isn't perfect!

Depending on what you want or need, validation can be very complicated, but since we have to start somewhere, we'll start with something simple - a name field on a web form.

For this example, we are only going to deal with "Western" names - we won't worry about dealing with Japanese glyphs and other scary-looking-Foreign characters.

First, we need to think what users might want to enter in this field. Secondly we need to think what evil shifty-eyed people might like to put in to break the site.

What users want.
Most usernames contain letters (upper and lower case), numbers, hyphens and underscores. If that was all we wanted, we could get away with the following regex:

```
[\w-]+
```

If you remember from the previous guide, \w is a shortcut for alphanumeric plus underscores. Since we want hyphens too, we added that to the character class.
If we don't trust shortcuts, we could have written it with ranges instead:

```
[a-zA-Z0-9_-]+
```

I tend to use the latter style - I find it easier to know what I'm matching. \w isn't terribly memorable. It's a little more verbose, and I never trusted all languages to have the same shortcuts, so I have stuck to using ranges of characters.
When you first write a regex to validate input, you have to start thinking of all validate input it could invalidate. The first thing that springs to mind is spaces. In the comments on the previous article, quite a few people used "Firstname Lastname" - perfectly valid input, but wouldn't pass the previous regex. Easily fixed, just add a space to the character-class:

```
[a-zA-Z0-9-_\ ]+
```

I have used an escaped-space instead of \s because I don't want to allow tabs. Also, some regex implementations don't require you to escape the space, others do.
For example, Python regex compiled with the re.VERBOSE flag will ignore whitespace and line-breaks; if you want to match a space, you either escape the space, or use \s.
Personally, I escape spaces them most of the time (since it works whether it ignores spaces or not), and I find it easier to spot where I want to match spaces.

Great, now we have a regex we can validate user input.. But not until we consider what evil people might put in to the field to break things..

What evil people want: SQL injections
There are two big problems when it comes to user-submitted data, which is going to be displayed on the site, SQL injections and Cross-site scripting attacks (XSS).

SQL injections are basically badly validated user-input being append to an SQL query. I won't go into much detail about them here. The main cause of these are un-escaped quotes in user-input. For example, here is a simple SQL query:

```
SELECT user_id FROM Students WHERE username='Robort'
```

Now, bob is what the user entered - but what if a user entered the following as a username:

```
Robort'); DROP TABLE Students;--
```

Bad things happen, if you don't validate their input.. The query turns into the following two queries:

```
     SELECT user_id FROM Students WHERE username='Robort'); DROP
TABLE Students;--'
```

If you’re not familiar with SQL, basically that went from retrieving the ID of a supplied username, to retrieving the username and deleting every single student record.

For more information, see the Wikipedia page on SQL injections: http://en.wikipedia.org/wiki/SQL_injection

So, how do we prevent this?
In the case of out name-field we’ve prevented this by not allowing quotes, brackets or semicolons to every make it near the SQL query.
What if we did need these symbols in the field? For example, quote marks are perfectly valid in a comment’s text - I just used one as an apostrophe.
We simply escape the quotes, by putting a backslash before them:

```
quote marks are perfectly valid in a comment\’s text
```

Most languages will have built in commands to escape quotes and such, and some database interfaces have this too - but what if our language doesn’t have such a command or feature? Regex can do that!

```
s/’/\\’/g
```

Looks rather horrible, but that’s the syntax for a regex-substitution in Perl.
The ‘s’ says it’s a substitution, then, between the first and second forward-slash is the pattern to look for, and between the second and last forward-slash is what to replace it with.
In this case, we look for a single quote, and replace it with backslash followed by a single quote. We need two backslashes, a single backslash would escape the next character, two gives a literal backslash.
Finally, the ‘g’ is a modifier, that tells the regex to be applier to every occurrence of the quote, not just the first.
If you use the above regex, you must first escape entered backslashes (\ ), then single quotes (‘), double quotes (“) and backticks (`). Be very careful and test thoroughly!

What evil people want 2: Cross-Site-Scripting Attacks
The second big problem is cross-site-scripting attacks. An XSS attack is injecting extra HTML onto a page.
These attacks can range from slightly annoying (a Javascript alert saying “hah”), to having users accounts compromised.

Typically these are Javascript <script></script> tag containing some code to take the users authentication cookie, and send it to the attacker - allowing them to log in as the user.
There was a big XSS attack on Myspace, which acted as a worm - when someone viewed an “infected” profile page, it would submit a friend-request to it’s creator, then add the XSS-code to the users profile. It ended up on over a million users profiles, all because the sites creators accidently allowed Javascript to be entered in one small part of the users profile! Read more about it on the creators website: http://namb.la/popular/

The problem is allowing angle-brackets (< and >), then displaying them back on the page without converting them to &lt; and &gt;
If you don’t this, they get interpreted as regular HTML. Using CSS, you can “deface” a website by creating a div, and making it float above the website. Using javascript you can redirect to another website, or steal the users cookie that automatically logs them in.

As with SQL injections, most languages have functions to escape HTML entities, or alternatively you can achieve this with regular expressions:

```
s/</&lt;/g
s/>/&gt;/g
```

If you want to read more on XSS attacks, see the Wikipedia page: http://en.wikipedia.org/wiki/Cross-site_scripting

Back to the name validation

So we know users cannot inject SQL into our query, as we are not allowing quotes in the input. Same with XSS injections, we are not allowing any characters that could be used to inject any extra HTML.

There is one last problem that might come up - input length.
Typically in your database, you have a VARCHAR row named “username” or similar, limited to around 30 characters. If the user inputs a 100 character name, it will get input happily, but it will be truncated with no warning.

A 100 character name will be happily (and wrongly) be validated by our regex:

```
[a-zA-Z0-9-_\ ]+
```

Remember, that plus symbol means one or more characters from this set of characters. That means we can have a million letter a’s and will validate.
We need to only allow one to thirty characters - the {1,30} quantifier seems perfect for this.

```
[a-zA-Z0-9-_\ ]{1,30}
```

It now reads: 1-to-30 characters from this set. It will not validate if there are invalid characters, or if it is blank or too long.

Done yet?

We now have a fairly issue-free regular expression to validate the input from a “Your name” field on a web-form or other input, but there are still issues.
A user could simply enter a single space, and it will be valid. This is not a desirable action.

The best way to fix this would be to trim trailing spaces from the input, then run it through our regex. As with escaping quotes and HTML, many languages have such a trim() function, and again, we can do this in regular expressions:

```
s/\s+$//
```

Substitute one or more spaces directly before the end of the string.

The dollar sign ($) means the end of the string, and the carat (^). To match the word “hi” at the start of a string only, you would do:

```
^hi
```

And at the end of a string:

```
hi$
```

And you can combine the two, for example to match only the word hi, with nothing else in the string:

```
^hi$
```

To match a sentence, starting with the word hi, and ending in bye:

```
^hi.*bye$
```

Those two symbols are very useful, and I really should have mentioned them earlier!
So now if a user enters only spaces, they are all stripped away, making the input 0 characters long, so are invalidated by {1,30}

So.. we're done? I'm afraid not! What if a user types 'a   b', that's valid, it's within [a-z\ ]{1,30}. Again, we deal with this the same way as the only-spaces problem: strip more than one space in a row, then run it though our main regex:

```
s/\s{2,}/ /
```

This replaces two or more spaces with a single space.

Lets leave it at that..
As you can probably see, validation can get very complicated. Even a seemingly simple thing can have problems - either invalidating otherwise valid input, or allowing malicious data to be entered.
Generally, it's best to err on the side of caution and limit the character set to a-zA-Z0-9, rather than accidently allowing the user to input extra commands or code into your site.

Make sure you let users know what restrictions there are on inputs (For example, "Usernames must contain only upper or lower case letters, numbers, underscores and hyphens"), and instead of simply saying "Your form was filled in invalidly", remember to let users know what they did wrong (highlight the invalid field, remind them of the allowed characters)

Extracting text from a string

This is something I use constantly. Say you have a HTML file, and you wish to get all the URLs within the file. Without using regular expressions, you are stuck using a horrible temperamental mess of strpos(), substr() and such commands. With regular expressions, you can find all images in a single expression.

```
<img.*? src=”(.+?)”.*?>
```

Then you use your languages regex match-all function, and you will be presented with an array of all images.

Hopefully by this point you can work out what most of that regex does. First we look for the string <img, then look for the src=”” attribute, grabbing the contents of it using a group (the match within the brackets), then we look for the closing tag.

Greedy Regexs
This is a good time to explain greedy/non-greedy matching. Greedy matching would be to simply do

```
<img.* src=”.+”.*>
```

While this might work work, it can behave very strangely when there are multiple double-quotes, or multiple tags.
Trying to match <.*> will find the first angle-bracket, and then the last one - this is greedy matching. While this behavior is sometimes desired, other times it can make the regex act strangely.
What we want it to do is find the next closing-angle-bracket, so we use non-greedy matching:

```
<.*?>
```

The question mark after a quantifier indicates non-greedy matching:

```
.*?
.+?
.{1,4}?
```

There is another other way to do achieve non-greedy matching:

```
< [^>]+ >
```

That looks for <, then one or more characters that are not >, and finally the >
For matching things like HTML tags, it is more reliable, and you don’t have to worry about greedy/non-greedy quantifiers.

So, our img-src grabbing regex updated to use the [^>]+ syntax

```
<img.*? src=”([^”]+)”[^>]+>
```

Like all regular expressions, this one has a problem - HTML can use either single or double quotes. We can use the alternation symbol, the vertical pipe (I), which you will be familiar with as the boolean OR in many programming languages.

As a simple demonstration of alternation, I want to be able to match the word hi, or bye, followed by an exclamation mark or a question mark:

```
(hi|bye)[!?]{1}
```

That matches hi OR bye, followed by one character from a set of [!?]. It could be written using an alternation for the ! and ?, but to match a single character, it’s far neater to use a character class and match one symbol (we can’t use the ? as it matches “zero or one”). You don’t need to escape the ? within a character class, as it has no meaning there. Were we to use alternation, it would require escaping.

To update our img-src grabbing regex to allow both single and double quotes

```
<img.*? src=[“‘]{1}([^”’]+)[”’]{1}[^>]+>
```

I’m aware that looks horrible, but it was easy for me to write. You don’t write a regular expression in a single go - you start with a basic match, and refine it as you notice problems. This is why reading or editing other users regular expressions is so difficult.

Groups

Now, somewhere amidst that line-noise is a group - a match enclosed in brackets. For the sake of readability, I’ll revert back to the simpler looking regex (before we dealt with the single/double quote issue)

```
src=”(.+?)”
```

This matches src=”, then stores everything that matches .+ until the next “

This is best explained via an example:

```
Input:    <img src=”test.png” alt=”Test picture”>
Regex:    <img src=”(.+?)” alt=”(.+?)”>
Group 1:  test.png
Group 2:  Test picture
```

I’m not even going to get into the problems with that regular expression! I will explain how to grab those groups in various languages in the next section, I’ve just one thing to explain, back-references.

Back-references are one of the things I put off learning because they sounded scary - but just like regular expressions, they are easier than they seem. Simply, you can use group-matches within the same regular expression. To be honest, you won’t use these too often. The only reason I’ve used them was to match HTML tags:

```
<(.+?)>.+?</\1>
```

Not the scariest looking regex. That finds the first <tag>, storing “tag” in group 1. Then looks for </ group-1 > (in this case it would be </tag>)

Regex in Languages

Now, the bit you've possibly been waiting for - how to use simple regular expressions in various languages.

In these examples, we will start by storing a string in a variable, then we will apply the regular expression. For matching, we will display a message if it was successful, for substitutions we will display the modified variable, and for extraction we and display the groups.

These examples will use a simple non-greedy match (For example: /st.+?g/), but we could use the more complicated character-classes, back-references, or anything else we have discussed previously.

Also, in many languages, to improve readability, you can use symbols other than a forward-slash to indicate start/end of a regex - you can use various characters, for example:

```
$strings=~s#'#\\'#g
$strings=~s^'^\\'^g
```

Many, many languages have built-in regex capabilities, there's no way I can cover them all, so I will focus on a few languages I consider "the biggest" for readers of an article like this. As I have said previously, the Perl regular-expressions syntax is the most common, but there are variations - as always, check that languages documentation, or have a look around for tutorials and guides.

Perl

Since we are using Perl Compatible Regexs, Perl seems like a sensible place to start.

Simple matching

```
my $string = “Test string!”;
if( $string=~/st.+?g/ ){
     print “Match”;
}
```

Substitution

```
my $string = “Test string!”;
$string=~s/st.+?g/replacement/;

print ($string);
```

Extraction

Using default variables:

```
my $string = "<img src='test.png' alt='Test picture'>";
$string=~/<img src='([^']+)' alt='([^']+)'/;

print ($1 . "\n");
print ($2 . "\n");
```

Storing groups to named variables:

```
my $string = "<img src='test.png' alt='Test picture'>";
my ($match_one,$match_two) = $string=~/<img src='([^']+)'
alt='([^']+)'/;

print ($match_one . "\n");
print ($match_two . "\n");
```

Ruby

Ruby regex handling is very similar to Perl. I don’t have much experience with Ruby, so there may be better ways of doing this - these examples work, and should be enough to get you started. If you want to learn more about Ruby regex, I recommend reading the Ruby documentation, or the many online tutorials.

Simple matching

```
string = “Test string!”;
if string =~ /st.+?g!/
    puts “Match”
end
```

Substitution

```
string = “Test string!”
string = string.gsub /st.+?g/, “replacement”

puts string
```

Extraction

Using default variables:

```
string = "<img src='test.png' alt='Test picture'>"
string.scan /<img src='(.+?)' alt='(.+?)'>/

puts $1
puts $2
```

Storing groups to named variables:

```
string = "<img src='test.png' alt='Test picture'>"
match_1,match_2 = /<img src='(.+?)' alt='(.
+?)'>/.match(string).to_a

puts match_1
puts match_2
```

Python

Regex in python is a module called “re”. There are several different regular expression functions, some more suitable to certain situations. For example, match() will match the pattern from the start of the string, whereas search() will match anywhere in the string.

Simple matching

```
import re
string = "Test string!"
if re.search("st.+?g", string):
     print "Match"
```

Substitution

```
import re
string = "Test string!"
string = re.sub("st.+?g", "replacement", string)

print string
```

Extraction

Using default variables:

```
import re
string = "<img src='test.png' alt='Test picture'>"
results = re.search("<img src='(.+?)' alt='(.+?)'>", string)

print results.group(1)
print results.group(2)
```

Storing groups to named variables:

```
import re
string = "<img src='test.png' alt='Test picture'>"
match_1, match_2 = re.search("<img src='(.+?)' alt='(.+?)'>",
string).groups()

print match_1
print match_2
```

PHP

Regex in PHP is a little different to the other languages, being similar in syntax to C. The functions start with preg_, which stands for Perl Reg(ular expressions), so obviously it uses Perl-compatible regexs.

Simple matching

```
<?
$string = “Test string!”;
if( preg_match(“/st.+?g/”, $string) ){
     echo(“Match”);
}
?>
```

Substitution

```
<?
$string = “Test string!”;
$string = preg_replace(“ st.+?g”,”replacement”, $string);

echo($string);
?>
```

Extraction

PHP is a little different to the other languages - in the third argument, you supply a variable which receives the results.

```
<?
$string = "<img src='test.png' alt='Test picture'>";
preg_match_all("/<img src='(.+?)' alt='(.+?)'>/", $string,
$results);

echo($results[1][0] . "\n");
echo($results[1][0] . "\n");
?>
```

Javascript

Simple matching

```
string = "Test string!";
if( string.match(/st.+?g/) ){
     alert("Match");
}
```

Substitution

```
string = "Test string!";
string = string.replace(/st.+?g/, "replacement");
alert(string);
```

Extraction

```
string = "<img src='test.png' alt='Test picture'>";
string.search( /<img src='(.+?)' alt='(.+?)'>/ );

alert(RegExp.$1);
alert(RegExp.$2);
```

Fineto

We've covered some practical examples, and gone over a bit more regular expression syntax, then explained how to do the basic regular expression stuff in a range of programming languages.
Even though this guide is over 3000 words, I have only covered a small percentage of what you can do with regular expressions - but I hope this has helped make the regex syntax look a bit less like line-noise to you.

One final tip: Many times I've had a regular expression that didn't work for some reason. Since regexs are so difficult to read, and the order of characters is so important - a missing character or two in the wrong order can completely break a match, and mistakes can be very hard to spot.
More often that not, it is easier just to rewrite the expression than to spend hours trying to debug them.

If you notice any problems in the guide, or you have any tips or advice, leave a comment!

If you like this guide, Digg the story, submit it to Del.icio.us, upmod it on Reddit, send the link to your friends, and subscribe to the RSS feed!

Links

RSS feed:
http://neverfear.org/rss-full.rss
The article on neverfear.org:
http://neverfear.org/blog/view/
Regex_tutorial_for_people_who_should_know_Regex__but_do_not___Part_2/