

ByteNoise

A Hacker's Guide to Regular Expressions

So what are they?

Regular expressions are a powerful way to search for text that matches a certain criterion, and optionally replace it or parts of it with other text. They are supported (at least, distinct variations, commonly known as different "flavours," are supported) by many different languages and programs, so after only slight tweaking, the regular expression you use in your PHP code can be used in vi. In this beginner's guide, I'll only cover searching, but even this should be enough to give you a glimpse of how versatile regular expressions are.

The main part of the search

You can search for any standard phrase using regular expressions. This is the most basic way of using a regular expressions based program such as egrep, but can still be useful. For example, searching a list of animals for `lion` will bring up the following results:

```
lion  
lioness  
stallion
```

Note that it doesn't just search for that word, but any line containing the characters you specified in the right order. When using regular expressions in this way, all you have to remember is to escape any metacharacters (any characters that the program should not take literally) with a backslash.

Metacharacters: `\ / . ^ $? * + { } [] (|)`

As you have to escape any other metacharacters with a backslash, the backslash itself is also a metacharacter. If you wanted to search for the phrase `and\or`, you would need to type in `and\\or`. This applies to all the other metacharacters as well, so if you wanted to search for `$10`, you would need to type in `\$10`. Now you can search for any literal string of characters.

It's generally a good idea to let computers do the boring, repetitive work for you, so let's see what the metacharacters can do to make your life even easier. First is the dot, which matches any character. Try searching for `mo.se`. Your computer will find these matches:

```
moose  
mouse
```

You can use as many of these as you like. Typing in `.om.at` will give you the following:

```
tomcat  
wombat
```

If you remember what I said earlier about regular expressions only matching part of a line, you might think that the first dot is unnecessary, as either way, anything can come before the letter o. You'd be right. There is a subtle difference, however: putting a dot there means that there must be at least one character before the o, even though it can be anything. If any line that would otherwise match began with the o itself, it wouldn't count.

Another useful metacharacter is the caret (^), which means "the beginning of the line." If you search for `^lion` then your computer will include `lion` and `lioness` on the list, but not `stallion`. Similarly, the dollar sign means "the end of the line," so a search for `pig$` would give you `pig` and `guineapig`, but not `pigeon`.

You can use these together in any combination (as long as the caret only appears at the beginning of a line, and the dollar sign only appears at the end of it). For example, if you search for `^.at$`, your computer will give you the following words:

```
bat
cat
rat
```

The next five metacharacters are called quantifiers. They tell the program *how many* instances of the last character (or group of characters, but we'll get to that later) it should match. The question mark means "zero or one," the asterisk means "zero, one or more" and the plus symbol means "one or more."

A good use of the question mark is when you're searching text that could use either British or American spelling. If you wanted

to search for any instance of the word `flavour` or `flavor`, then you could combine them into a single search by typing in `flavou?r`. This means that the character directly before the question mark is optional, so both words will match.

It's worth noting that `.*` and `.+` will match any letters, not just one letter repeating several times. For example, using `egrep` to search for `^b.*bird$` will make it look for the following: the beginning of the line, the letter `b`, any number of any characters (including none), the letters `b`, `i`, `r`, `d`, then the end of the line. It will give you the following matching words:

```
blackbird  
bluebird
```

The braces (`{` and `}`) let you specify exactly how many characters you want to match. For example, `^l.{3}bird$` will match the beginning of a line, the letter `l`, any three characters, the letters `b`, `i`, `r`, `d`, then the end of the line. The following words match:

```
ladybird  
lovebird
```

This can be taken one step further by putting two numbers in the braces, separated by a comma. The first is the minimum number of times the character must be matched, and the last number is the maximum number of times. For instance, `Br{2,4}!` will match `Brr!`, `Brrr!` and `Brrrr!`.

Any one of these single characters

The square brackets are used to group single characters together. Say, for instance, that you want to look for `bat` and `cat` but not `rat`. You can use the regular expression `[bc]at` to specify that either `b` or `c`, but nothing else, can precede `a` and `t`.

This part of the regular expression is called a character class, and has two metacharacters of its own. This time, however, you don't need to use the backslash to escape them.

Metacharacters: `^` -

The first metacharacter in the character class is a carat. Although it usually means "the beginning of the line," here it means something else entirely. When placed at the very beginning of the character class, after the opening square bracket, it means that the following characters are the ones that must *not* appear. Searching for `[^bc]at`, for example, would find `rat` but not `bat` or `cat`. This is called a *negated* character class.

When placed at the beginning of the character class (or directly after the carat if it's a negated character class), the dash is literal. Otherwise, it is taken to mean "anything between these two characters." `[a-z]` matches any lowercase letter, `[A-Z]` matches any uppercase letter, and `[0-9]` matches any number. These can be combined. For example, `[A-Za-z]` matches any letter at all, while `[^A-Za-z]` matches any character that *isn't* a letter.

Character classes can also be combined with quantifiers, which

is where the fun really begins. For instance, you could use the regular expression `[aeiou]{5}` on a word list to find out which words contain five vowels in a row (which will return the word `queueing`).

Any one of these groups of characters

You can use normal brackets to specify a list of several groups of characters, any of which can be regarded as a match. These groups of characters are separated by the pipe symbol (`|`).

Metacharacters: `|`

Say that you want to find all instances of `blackbird` and `bluebird`, but no other birds beginning with the letter `b` that might be in the text you're searching. You can use the brackets and pipe symbol to make a list of exactly which groups of characters are allowed. The regular expression `(blackbird|bluebird)` would match just these two words, but there's a much more concise way of saying the same thing: `bl(ack|ue)bird`. This essentially tells the program exactly the same thing, but in a more efficient way. You can specify as many possibilities as you like, as long as they are grouped together by brackets and separated by pipes.

You can also use brackets to combine the grouped characters with a quantifier. This even works with a list of just one group of characters. For example, the regular expression `pig(eon)?` matches both `pig` and `pigeon`.

Putting it all together

Now you can put all of these ideas together. For a more geeky

example, let's say you're searching some old text files for any mention of the Commodore 64 computer. It's called several things, mainly the `Commodore 64`, `Commodore-64`, `Commodore64`, `C 64`, `C-64` and `C64`. It's possible that people might even have spelt it with a lowercase letter `c`.

To start with, you search for the letter `C`. As it can be either upper or lowercase, you use `[Cc]`. Next is the optional rest of the word, so you enclose the next characters in brackets to tell the program that they're to be treated as one entity, then use a question mark to indicate that this particular entity is optional: `(ommodore)?`. Either a space, a dash, or nothing at all comes next, so `[-]?` is the logical choice (remember to keep the dash to the left). Last of all, you are confident that people will use the digits `64`. Putting it all together gives you `[Cc](ommodore)?[-]?64`. It looks complex when it's assembled together, but hopefully it shouldn't be difficult to make in the first place. Just remember to comment your code so that when you come back to it later, or someone else inherits your code, it isn't too difficult to work out what's going on.

The next step

This is just a beginner's guide. Hopefully you should now have an appreciation of how useful regular expressions can be, and an appetite to learn more. A good first step is to download a free version of `grep` and a comprehensive word list. Setting yourself tasks like "find every word that contains all five vowels in order" can be an excellent way to practice your knowledge of regular expressions. In the longer term, a good book such as Jeffrey E. F. Friedl's *Mastering Regular Expressions* (published by the ubiquitous O'Reilly Media) can provide more in depth knowledge, including the particulars of each different flavour of

regular expressions.

Example regular expressions

Here are some example regular expressions to help you on your way:

Regular expression	Matches
<code>[-_a-zA-Z0-9]+@[-_a-zA-Z0-9.]+</code>	Any e-mail address
<code>alt(\\. [a-z0-9]+)+</code>	Any alt. newsgroup

The example wordlist

This is the wordlist used in my guide:

bat
blackbird
bluebird
cat
groundhog
guineapig
ladybird
lion
lioness
lovebird
moose
mouse
pig
pigeon
rabbit
rat
stallion
tomcat

wombat

Resources

- Comprehensive wordlists can be downloaded from <ftp://ftp.ox.ac.uk/pub/wordlists/>.
- GNU Grep can be downloaded from <http://www.gnu.org/software/grep/grep.html>.