Linux Fu: Regular Expressions

Originalartikel

Backup

<html> If you consider yourself a good cook, you may or may not know how to make a souffle or baklava. But there are certain things you probably do know how to do that form the basis of many recipes. For example, you can probably boil water, crack an egg, and brown meat. With Linux or Unix systems, you can make the same observation. You might not know how to set up a Wayland server or write a kernel module. But there are certain core skills like file manipulation and editing that will serve you no matter what you do. One of the pervasive skills that often gives people trouble is regular expressions. Many programs use these as a way to specify search patterns, usually in text strings such as files. If you aren’t comfortable with regular expressions, that’s easy to fix. They aren’t that hard to learn and there are some great tools to help you. Many tools use regular expressions and the core syntax is the same. The source of confusion is that the details beyond core syntax have variations. Let’s look at the foundation you need to understand regular expression well. <h2>Programs that Utilize Regexp</h2> Perhaps the most obvious program that uses regular expressions — or regexps, for short — is grep. This is a simple program that takes a regexp and one or more file names (unless you want to read from standard input). By default, it prints any lines that match the regexp. Like I said, simple, but powerful and one of the most used command line tools which is why I use it as the example in this article. But grep isn’t the only program that uses regular expressions. Awk, sed, Perl, editors like VIM and emacs, and many other programs can use regular expressions for pattern matching. And because regexp is so common you will see it pop up in user settings and even web apps as a way to extend the utility of the applications. Depending on your system, you may have similar programs like egrep. On my system, egrep is just a wrapper around grep that passes the -E option which changes the kind of regular expression grep parses. For the purpose of this post, I’ll talk about egrep by default. <h2>Character Matching and Classes</h2> Consider this command line: <pre class=„brush: plain; title: ; notranslate“ title=„“> egrep dog somefile.txt </pre> This would search somefile.txt and match the following lines: <pre class=„brush: plain; title: ; notranslate“ title=„“> I am a dog. There is a special dogma involved. —dog— </pre> <h3>Period as Wildcard</h3> It would not match “Dog” because regexps are case sensitive by default. If that was all there was to it, regexps would not be a very interesting topic. You can use a period as a sort of wildcard that will match any character. So the pattern

```
"d.g"
```

would match both “dog” and “dig” (or “d$g” for that matter). <h3>Escape Characters</h3> What happens if you want to match a period, then? You can escape any special character with a backslash. So

```
"d\.g"
```

will match “d.g” and nothing else. However, keep in mind the tool you are using to enter the regular expression might use backslashes, too. For example, consider this session: <img data-attachment-id=„295989“ data-permalink=„https://hackaday.com/2018/03/09/linux-fu-regular-expressions/grepesc/“ data-orig-file=„https://hackadaycom.files.wordpress.com/2018/02/grepesc.png?w=800&h=360“ data-orig-

size=„664,299“ data-comments-opened=„1“ data-image-meta=„{"aperture":"0","credit":"","camera":"","caption":"","created_timestamp":"0","copyright":"","focal_length":"0","iso":"0","shutter_speed":"0","title":"","orientation":"0"}“ data-image-title=„grepesc“ data-image-description=„“ data-medium-file=„https://hackadaycom.files.wordpress.com/2018/02/grepesc.png?w=800&h=360?w=400“ data-large-file=„https://hackadaycom.files.wordpress.com/2018/02/grepesc.png?w=800&h=360?w=664“ class=„aligncenter wp-image-295989“ src=„https://hackadaycom.files.wordpress.com/2018/02/grepesc.png?w=800&h=360“ alt=„“ width=„800“ height=„360“ srcset=„https://hackadaycom.files.wordpress.com/2018/02/grepesc.png 664w, https://hackadaycom.files.wordpress.com/2018/02/grepesc.png?w=250&h=113 250w, https://hackadaycom.files.wordpress.com/2018/02/grepesc.png?w=400&h=180 400w“ sizes=„(max-width: 800px) 100vw, 800px“/> There is a gotcha here. The shell interprets

```
\.
```

as a single period. It doesn’t see the backslash are part of the search but as an escape character in the shell itself. To properly pass

```
\.
```

to grep, you have to escape the backslash (

```
\\
```

) as in the second example. Quoting special characters can get tricky and depends on what shell you use. If you use bash, you might want to just enclose your expressions in quotes or double quotes. Even then, the rules about backslashes still apply. <h3>Character Classes</h3> Sometimes you don’t want a single character, but you don’t just want any character, either. That’s where character classes come in. For example: <pre class=„brush: plain; title: ; notranslate“ title=„“> egrep [XYZ][0-9][0-9][0-9]V afile.txt </pre> This will select strings like “X000V” or “Z123V”. It is common to use these to allow case insensitivity (for example,

```
[aA]
```

or

```
[a-zA-Z]
```

). You can also create a negative character class using the ^ character. For example,

```
[^XYZ]
```

will match any character except X, Y, or Z. The If you need to match a dash (-), it should come first. If you need to match a caret (^), it should not come first. Since the order of characters in a class

isn’t significant, that won’t cause problems. For example, if you want to match any digit, a dash, or a caret you can write: <pre class=„brush: plain; title: ; notranslate“ title=„“> egrep [-0-9^] </pre> <h2>Repeats</h2> Finding dog, dig, dug, and d$g is fine, but we need to be able to do more. A very powerful part of regular expressions is that you can specify that a character repeats. You can also make a character optional. For example here’s a match for bar or bear:

```
"be?ar"
```

 A very common requirement is to have zero or one occurrences of a pattern (that is, make it optional). You might also want zero or more characters repeating or, sometimes, one or more. Suppose you wanted to match a number in a file from a data logger with an optional negative sign and a decimal point. You might use

```
"-?[0-9]+\.[0-9]*"
```

. It may be helpful to understand this syntax if you to walk through it graphically in the diagram. <figure id=„attachment_298372“ class=„wp-caption aligncenter c2“><figcaption class=„wp-caption-text“>Diagram via Regexper CC-BY 3.0</figcaption></figure> That pattern will match -25.2 or 33. or 17.125. The plus sign indicates that there must be one digit, but there can be more. The * allows for any number of digits including none at all (that is, zero or more matches). Note the decimal point needs an escape. The funny part is, it will work without it because the period will match “any character.” The bad part about that is it will also match things like 14×2 which isn’t a properly formatted number. <h3>Getting Advanced with Repeats</h3> A more advanced regular expression repeat is to use curly braces to specify how many times a pattern can repeat. Some tools require you to escape the curly braces, but grep doesn’t. So to match exactly 4 lower case letters in a row you can use the pattern

```
"[a-z]{4}"
```

 Of course, that’s the same as saying

```
[a-z][a-z][a-z][a-z]
```

but easier to type. You can also set a lower and upper limit as in:

```
[a-z]{2,4}
```

which would match two to four letters. This is the same as

```
[a-z][a-z][a-z]?[a-z]?
```

 <h3>Multiple Match Behavior Varies by Program</h3> For grep, the fact that a match occurred is sufficient. For some other tools, it matters if you match the most characters that match or the least. For example,

```
"abc*"
```

applied to a string containing “abccccc” could match ab or it could match the whole thing. This depends on the tool (and sometimes options to the tool or command) but it is good to keep in mind. Different tools work differently, too, on how they handle multiple matches. For example, matching the pattern “X” with a string: “XyyX” could match the first X or both, it just depends. Again, for grep, it doesn’t matter. If the line matches at least once, that’s all that is important. <h2>Anchors</h2> You probably noticed that the pattern match can occur anywhere in the string. You can anchor the match to the start of the string with the ^ character and the end of the string with a $ character. For example, this matches blank lines:

```
"^ *$"
```

or if you want to include tabs:

```
"^[ \t]*$"
```

 As in C, the \t character is a tab. Depending on the tool, there may be other escape characters available, too. You can match lines that have a percent sign as the first nonblank character like this

```
"^[ \t]*%"
```

 <h2>Grouping</h2> There are a few ways to group regular expressions. You can use parenthesis, although some tools require you to escape them if you want to use them for grouping. For example: <pre class=„brush: plain; title: ; notranslate“ title=„“> egrep a(bc)?d </pre> <figure id=„attachment_298377“ class=„wp-caption alignright c3“><img data-attachment-id=„298377“
data-
permalink=„https://hackaday.com/2018/03/09/linux-fu-regular-expressions/regexp-grouping-png/“ data-orig-file=„https://hackadaycom.files.wordpress.com/2018/03/regexp-grouping.png“ data-orig-size=„492,192“ data-comments-opened=„1“ data-image-

meta=„{"aperture":"0","credit":"","camera":"","caption":"","created_timestamp":"0","copyright":"","focal_length":"0","iso":"0","shutter_speed":"0","title":"","orientation":"0"}“ data-image-title=„regexp-grouping.png“ data-image-description=„“ data-medium-file=„https://hackadaycom.files.wordpress.com/2018/03/regexp-grouping.png?w=400&h=156“
data-large-file=„https://hackadaycom.files.wordpress.com/2018/03/regexp-grouping.png?w=492“
class=„wp-image-298377 size-medium“
src=„https://hackadaycom.files.wordpress.com/2018/03/regexp-grouping.png?w=400&h=156“
alt=„“ width=„400“ height=„156“
srcset=„https://hackadaycom.files.wordpress.com/2018/03/regexp-grouping.png?w=400&h=156 400w, https://hackadaycom.files.wordpress.com/2018/03/regexp-grouping.png?w=250&h=98 250w, https://hackadaycom.files.wordpress.com/2018/03/regexp-grouping.png 492w“ sizes=„(max-width: 400px) 100vw, 400px“/><figcaption class=„wp-caption-text“>Image via Regexper CC-BY 3.0</figcaption></figure> This would look for a match starting with “a” then it will match a “d” or a “bcd”. This isn’t the same as

```
"ab?c?d"
```

because that would match “acd”. You can also use a pipe character as an “OR” operator. For example,

```
"a|b"
```

will match either a or b, which isn’t much different from

```
"[ab]"
```

 However, combined with grouping this is useful. For example,

```
"(dog)|(cat)"
```

will match either animal. In some tools, the grouping will mark a part you want to read as part of some other code or make available in a replacement. For example, in some tools, you might search

```
"id=([0-9])+"
```

then in the replacement string you could use \1 to refer to whatever number matched the expression in the parentheses. The exact details vary by tool. For example, some might use &1 or some other syntax. <h2>Motivation: How Is Regexp Useful?</h2> There’s more but let’s take a break and figure out what good all this is. To do that, assume you have a file full of data from a logger that has temperature data in degrees C. It also has other data in it. However, all the temperatures are in the format of a number followed by a space and an upper case C. So temperatures might be “-22 C” or “13.5 C” as examples. It would be easy to convert these numbers using a program like awk. I don’t want to get into too many details about awk, but it has a

```
match
```

function and a

```
gensub
```

function that find and replace regular expressions, respectively. It also allows you to filter lines using a rule that is just a regular expression between two slashes. Consider this: <pre class=„brush: plain; title: ; notranslate“ title=„“> / C$/ { if (match($0,/([-0-9.]+) C/,matchres)) # . inside [] doesn't need escape

```
  print gensub(/[-0-9.]+ C/,matchres[1]*9.0/5.0+32 " F","g")
else
  print
```

next; # start over with next line } { print } # print other lines </pre> The first line matches text that has a space and a C at the end of the line. The

```
match
```

subroutine gets the number in

```
matchres[1]
```

. Granted, it can parse nonsense like “-…C” but you can assume there’s nothing like that in there. You could do better with

```
"-?[0-9]\.?[0-9]*"
```

perhaps. The

```
gensub
```

function does the conversion. Plenty of regular expressions to make this work. <h2>Variations</h2> Unfortunately, there are subtle differences between different tools. [Donald Knuth] once said: <blockquote> I define UNIX as “30 definitions of regular expressions living under one roof”. </blockquote> I’ve mentioned some of these. For example, some implementations treat parenthesis as grouping unless they are escaped while others require you to escape them if you want them to group and not be regular characters. It is common to see shorthand expressions for things like digits and spaces. This is good because it ought to understand the system’s localization, too. For example

```
[:digit:]
```

is a stand-in for

```
[0-9]
```

. Some tools use special escape characters like \d for the same purpose. Instead of trying to describe it all, I’m going to simply direct you to a table

from Wikipedia that describes some common systems. <h2>Tools for Debugging Regexp</h2> That’s all you really need to know. If you want to debug regular expressions interactively. there are plenty of tools for that online. You can also get a visual diagram of a regular expression to help work your way through them. You might also enjoy a library of regular expressions you can — ahem — borrow. If you’d rather learn while having fun, you might try crosswords or even golf. </html>