Chapter 19. Regular expressions

Regular expressions are a very powerful tool in Linux. They can be used with a variety of programs like bash, vi, rename, grep, sed, and more.

This chapter introduces you to the basics of **regular expressions**.

19.1. regex versions

There are three different versions of regular expression syntax:

```
BRE: Basic Regular Expressions
ERE: Extended Regular Expressions
PRCE: Perl Regeular Expresseions
```

Depending on the tool being used, one or more of these syntaxes can be used.

For example the **grep** tool has the **-E** option to force a string to be read as ERE while **-G** forces BRE and **-P** forces PRCE.

Note that **grep** als has **-F** to force the string to be read literally. The **sed**

tool also has options to choose a regex syntax.

Read the manual of the tools you use!

19.2. grep

19.2.1. print lines matching a pattern

grep is a popular Linux tool to search for lines that match a certain pattern. Below are some examples of the simplest **regular expressions**.

This is the contents of the test file. This file contains three lines (or three **newline** characters).

```
paul@rhel65:~$ cat names
Tania
Laura
Valentina
```

When **grepping** for a single character, only the lines containing that character are returned.

```
paul@rhel65:~$ grep u names
Laura
paul@rhel65:~$ grep e names
Valentina
paul@rhel65:~$ grep i names
Tania
Valentina
```

The pattern matching in this example should be very straightforward; if the given character occurs on a line, then **grep** will return that line.

19.2.2. concatenating characters

Two concatenated characters will have to be concatenated in the same way to have a match.

This example demonstrates that **ia** will match Tan**ia** but not V**a**lent**i**na and **in** will match Valent**in**a but not Ta**n**i**a**.

```
paul@rhel65:~$ grep a names
Tania
Laura
Valentina
paul@rhel65:~$ grep ia names
Tania
paul@rhel65:~$ grep in names
Valentina
```

19.2.3. one or the other

PRCE and ERE both use the pipe symbol to signify OR. In this example we **grep** for lines containing the letter i or the letter a.

```
paul@debian7:~$ cat list
Tania
Laura
paul@debian7:~$ grep -E 'i|a' list
Tania
Laura
```

Note that we use the **-E** switch of grep to force interpretation of our string as an ERE.

We need to **escape** the pipe symbol in a BRE to get the same logical OR.

```
paul@debian7:~$ grep -G 'i|a' list
paul@debian7:~$ grep -G 'i\|a' list
Tania
Laura
```

19.2.4. one or more

The * signifies zero, one or more occurrences of the previous and the **+** signifies one or more of the previous.

```
paul@debian7:~$ cat list2
ll
lol
lool
loool
paul@debian7:~$ grep -E 'o*' list2
ll
lol
lool
loool
paul@debian7:~$ grep -E 'o+' list2
lol
lool
loool
paul@debian7:~$
```

19.2.5. match the end of a string

For the following examples, we will use this file.

```
paul@debian7:~$ cat names
Tania
Laura
Valentina
Fleur
Floor
```

The two examples below show how to use the **dollar character** to match the end of a string.

```
paul@debian7:~$ grep a$ names
Tania
Laura
Valentina
paul@debian7:~$ grep r$ names
Fleur
Floor
```

19.2.6. match the start of a string

The **caret character (^)** will match a string at the start (or the beginning) of a line.

Given the same file as above, here are two examples.

```
paul@debian7:~$ grep ^Val names
Valentina
paul@debian7:~$ grep ^F names
Fleur
Floor
```

Both the dollar sign and the little hat are called **anchors** in a regex.

19.2.7. separating words

Regular expressions use a **\b** sequence to reference a word separator. Take for example this file:

```
paul@debian7:~$ cat text
The governer is governing.
The winter is over.
Can you get over there?
```

Simply grepping for **over** will give too many results.

```
paul@debian7:~$ grep over text
The governer is governing.
The winter is over.
Can you get over there?
```

Surrounding the searched word with spaces is not a good solution (because other characters can be word separators). This screenshot below shows how to use **\b** to find only the searched word:

```
paul@debian7:~$ grep '\bover\b' text
The winter is over.
Can you get over there?
paul@debian7:~$
```

Note that **grep** also has a **-w** option to grep for words.

```
paul@debian7:~$ cat text
The governer is governing.
The winter is over.
Can you get over there?
paul@debian7:~$ grep -w over text
The winter is over.
Can you get over there?
paul@debian7:~$
```

19.2.8. grep features

Sometimes it is easier to combine a simple regex with **grep** options, than it is to write a more complex regex. These options where discussed before:

```
grep -i
grep -v
grep -w
grep -A5
grep -B5
grep -C5
```

19.2.9. preventing shell expansion of a regex

The dollar sign is a special character, both for the regex and also for the shell (remember variables and embedded shells). Therefore it is advised to always quote the regex, this prevents shell expansion.

```
paul@debian7:~$ grep 'r$' names
Fleur
Floor
```

19.3. rename

19.3.1. the rename command

On Debian Linux the **/usr/bin/rename** command is a link to **/usr/bin/prename** installed by the **perl** package.

```
paul@pi ~ $ dpkg -S $(readlink -f $(which rename))
perl: /usr/bin/prename
```

Red Hat derived systems do not install the same **rename** command, so this section does not describe **rename** on Red Hat (unless you copy the perl script manually).

There is often confusion on the internet about the rename command because solutions that work fine in Debian (and Ubuntu, xubuntu, Mint, ...) cannot be used in Red Hat (and CentOS, Fedora, ...).

19.3.2. perl

The **rename** command is actually a perl script that uses **perl regular expressions**. The complete manual for these can be found by typing **perldoc perlrequick** (after installing **perldoc**).

```
root@pi:~# aptitude install perl-doc
The following NEW packages will be installed:
  perl-doc
0 packages upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 8,170 kB of archives. After unpacking 13.2 MB will be used.
Get: 1 http://mirrordirector.raspbian.org/raspbian/ wheezy/main perl-do...
Fetched 8,170 kB in 19s (412 kB/s)
Selecting previously unselected package perl-doc.
(Reading database ... 67121 files and directories currently installed.)
Unpacking perl-doc (from .../perl-doc_5.14.2-21+rpi2_all.deb) ...
Adding 'diversion of /usr/bin/perldoc to /usr/bin/perldoc.stub by perl-doc'
Processing triggers for man-db ...
Setting up perl-doc (5.14.2-21+rpi2) ...
```

19.3.3. well known syntax

The most common use of the **rename** is to search for filenames matching a certain **string** and replacing this string with an **other string**.

This is often presented as **s/string/other string/** as seen in this example:

```
paul@pi ~ $ ls
abc       allfiles.TXT  bllfiles.TXT  Scratch   tennis2.TX
abc.conf  backup        cllfiles.TXT  temp.TXT  T
paul@pi ~ $ rename 's/TXT/text/' *
paul@pi ~ $ ls
abc       allfiles.text  bllfiles.text  Scratch    tennis2.text
abc.conf  backup         cllfiles.text  temp.text  tennis.text
```

And here is another example that uses **rename** with the well know syntax to change the extensions of the same files once more:

```
paul@pi ~ $ ls
abc       allfiles.text  bllfiles.text  Scratch    tennis2.text
abc.conf  backup         cllfiles.text  temp.text  tennis.text
paul@pi ~ $ rename 's/text/txt/' *.text
paul@pi ~ $ ls
abc       allfiles.txt  bllfiles.txt  Scratch   tennis2.txt
abc.conf  backup        cllfiles.txt  temp.txt  tennis.txt
paul@pi ~ $
```

These two examples appear to work because the strings we used only exist at the end of the filename. Remember that file extensions have no meaning in the bash shell.

The next example shows what can go wrong with this syntax.

```
paul@pi ~ $ touch atxt.txt
paul@pi ~ $ rename 's/txt/problem/' atxt.txt
paul@pi ~ $ ls
abc
abc.conf  allfiles.txt  backup        cllfiles.txt  temp.txt     tennis.tx
          aproblem.txt  bllfiles.txt  Scratch       tennis2.tx
paul@pi ~ $
```

Only the first occurrence of the searched string is replaced.

19.3.4. a global replace

The syntax used in the previous example can be described as **s/regex/replacement/**. This is simple and straightforward, you enter a **regex** between the first two slashes and a **replacement string** between the last two.

This example expands this syntax only a little, by adding a **modifier**.

```
paul@pi ~ $ rename -n 's/TXT/txt/g' aTXT.TXT
aTXT.TXT renamed as atxt.txt
paul@pi ~ $
```

The syntax we use now can be described as **s/regex/replacement/g** where s signifies **switch** and g stands for **global**.

Note that this example used the **-n** switch to show what is being done (instead of actually renaming the file).

19.3.5. case insensitive replace

Another **modifier** that can be useful is **i**. this example shows how to replace a case insensitive string with another string.

```
paul@debian7:~/files$ ls
file1.text  file2.TEXT  file3.txt
paul@debian7:~/files$ rename 's/.text/.txt/i' *
paul@debian7:~/files$ ls
file1.txt  file2.txt  file3.txt
paul@debian7:~/files$
```

19.3.6. renaming extensions

Command line Linux has no knowledge of MS-DOS like extensions, but many end users and graphical application do use them.

Here is an example on how to use **rename** to only rename the file extension. It uses the dollar sign to mark the ending of the filename.

```
paul@pi ~ $ ls *.txt
allfiles.txt  bllfiles.txt  cllfiles.txt  really.txt.txt  temp.txt  tennis.txt
paul@pi ~ $ rename 's/.txt$/.TXT/' *.txt
paul@pi ~ $ ls *.TXT
allfiles.TXT  aTXT.TXT    bllfiles.TXT    cllfiles.TXT    really.txt.TXT
temp.TXT      tennis.TXT
paul@pi ~ $
```

Note that the **dollar sign** in the regex means **at the end**. Without the dollar sign this command would fail on the really.txt.txt file.

19.4. sed

19.4.1. stream editor

The **stream editor** or short **sed** uses **regex** for stream editing. In this

example **sed** is used to replace a string.

```
echo Sunday | sed 's/Sun/Mon/'
Monday
```

The slashes can be replaced by a couple of other characters, which can be handy in some cases to improve readability.

```
echo Sunday | sed 's:Sun:Mon:'
Monday
echo Sunday | sed 's_Sun_Mon_'
Monday
echo Sunday | sed 's|Sun|Mon|'
Monday
```

19.4.2. interactive editor

While **sed** is meant to be used in a stream, it can also be used interactively on a file.

```
paul@debian7:~/files$ echo Sunday > today
paul@debian7:~/files$ cat today
Sunday
paul@debian7:~/files$ sed -i 's/Sun/Mon/' today
paul@debian7:~/files$ cat today
Monday
```

19.4.3. simple back referencing

The **ampersand** character can be used to reference the searched (and found) string. In this example the **ampersand** is used to double the occurrence of the found string.

```
echo Sunday | sed 's/Sun/&&/'
SunSunday
echo Sunday | sed 's/day/&&/'
Sundayday
```

19.4.4. back referencing

Parentheses (often called round brackets) are used to group sections of the regex so they can be referenced later.

Consider this simple example:

```
paul@debian7:~$ echo Sunday | sed 's_\(Sun\)_\1ny_'
Sunnyday
paul@debian7:~$ echo Sunday | sed 's_\(Sun\)_\1ny \1_'
Sunny Sunday
```

19.4.5. a dot for any character

In a **regex** a simple dot can signify any character.

```
paul@debian7:~$ echo 2014-04-01 | sed 's/....-..-../YYYY-MM-DD/'
YYYY-MM-DD
paul@debian7:~$ echo abcd-ef-gh | sed 's/....-..-../YYYY-MM-DD/'
YYYY-MM-DD
```

19.4.6. multiple back referencing

When more than one pair of **parentheses** is used, each of them can be referenced separately by consecutive numbers.

```
paul@debian7:~$ echo 2014-04-01 | sed 's/\(....\)-\(..\)-\(..\)/\1+\2+\3/'
2014+04+01
paul@debian7:~$ echo 2014-04-01 | sed 's/\(....\)-\(..\)-\(..\)/\3:\2:\1/'
01:04:2014
```

This feature is called **grouping**.

19.4.7. white space

The **\s** can refer to white space such as a space or a tab.

This example looks for white spaces (\s) globally and replaces them with 1 space.

```
paul@debian7:~$ echo -e 'today\tis\twarm'
today is warm
paul@debian7:~$ echo -e 'today\tis\twarm' | sed 's_\s_ _g'
today is warm
```

19.4.8. optional occurrence

A question mark signifies an the previous is **optional**.

The example below searches for three consecutive letter o, but the third o is optional.

```
paul@debian7:~$ cat list2
ll
lol
lool
loool
paul@debian7:~$ grep -E 'ooo?' list2
lool
loool
paul@debian7:~$ cat list2 | sed 's/ooo\?/A/'
ll
lol
lAl
```

19.4.9. exactly n times

You can demand an exact number of times the previous has to occur. This

example wants exactly three o's.

```
paul@debian7:~$ cat list2
ll
lol
lool
loool
paul@debian7:~$ grep -E 'o{3}' list2
loool
paul@debian7:~$ cat list2 | sed 's/o\{3\}/A/'
ll
lol
lool
lAl
paul@debian7:~$
```

19.4.10. between n and m times

And here we demand exactly from minimum 2 to maximum 3 times.

```
paul@debian7:~$ cat list2
ll
lol
lool
loool
paul@debian7:~$ grep -E 'o{2,3}' list2
lool
loool
paul@debian7:~$ grep 'o\{2,3\}' list2
lool
loool
paul@debian7:~$ cat list2 | sed 's/o\{2,3\}/A/'
ll
lol
lAl
lAl
paul@debian7:~$
```

19.5. bash history

The **bash shell** can also interpret some regular expressions.

This example shows how to manipulate the exclamation mask history feature from the bash shell.

```
paul@debian7:~$ mkdir hist
paul@debian7:~$ cd hist/
paul@debian7:~/hist$ touch file1 file2 file3
paul@debian7:~/hist$ ls -l file1
-rw-r--r-- 1 paul paul 0 Apr 15 22:07 file1
paul@debian7:~/hist$ !l
ls -l file1
-rw-r--r-- 1 paul paul 0 Apr 15 22:07 file1
paul@debian7:~/hist$ !l:s/1/3
ls -l file3
-rw-r--r-- 1 paul paul 0 Apr 15 22:07 file3
```

This also works with the history numbers in bash.

```
paul@debian7:~/hist$ history 6
 2089  mkdir hist
 2090  cd hist/
 2091  touch file1 file2 file3
 2092  ls -l file1
 2093  ls -l file3
 2094  history 6
paul@debian7:~/hist$ !2092
ls -l file1
-rw-r--r-- 1 paul paul 0 Apr 15 22:07 file1
paul@debian7:~/hist$ !2092:s/1/2
ls -l file2
-rw-r--r-- 1 paul paul 0 Apr 15 22:07 file2
```